Queue Machines

Bruno R. Preiss, Member, IEEE

Abstract—**Two formal execution models that use a queue as the underlying data structure for the manipulation of operands and results are presented. They are the ‘‘simple queue machine’’ and the ‘‘indexed queue machine.’’ An algorithm for generating the instruction sequence for evaluating an arbitrary expression on a simple queue machine is presented. It is shown that simple queue machines can efficiently exploit pipelined ALUs and support efficient concurrent expression evaluation.**

Acyclic dataflow graphs are shown to naturally lead to the generation of instruction sequences for the indexed queue machine execution model. Since queue machines are statically scheduled, they can exploit the techniques of conventional, Von Neumann architectures such as instruction look-ahead and overlapped execution of sequential instructions, thus overcoming a shortcoming of dataflow execution models. Practical indexed queue machines can be implemented by using registers that act as a cache for the front of the operand queue.

Index Terms—**Dataflow graphs, expression parse trees, instruction sequence generation, pipelined processing elements, queue machines.**

This research was supported in part by NSERC (Canada) under Grant OGP36635.
B. R. Preiss is with the Department of Electrical and Computer Engineering, University of Waterloo, Waterloo, Ontario, Canada, N2L 3G1.

Copyright (c) 1987 by Bruno R. Preiss. All rights reserved.

I. INTRODUCTION

Dataflow graphs are used to represent programs for dataflow machines[1]. Such graphs exhibit the concurrency inherent in a computation. Most dataflow machines attempt to detect and exploit this concurrency dynamically. However, the overhead associated with the dynamic concurrency detection and the inability of dataflow machines to exploit instruction look-ahead and overlapped sequential instruction execution prevent dataflow machines from realizing their full potential[2].

Many dataflow machines have been either proposed or built[3-6]. A common element of the organization of all these machines is a circular pipeline around which operands and results circulate. An important feature of such circular pipelines is the presence of a hardware queue that controls the flow of operands and results. In this paper, we introduce two execution mechanisms that explicitly use a queue as the underlying mechanism for the manipulation of operands and results. These execution models use static instruction schedules derived from acyclic dataflow graphs. By using static scheduling, it becomes possible to exploit the techniques of instruction look-ahead and overlapped sequential execution to achieve the efficient evaluation of acyclic dataflow graphs, thus overcoming a shortcoming of previous dataflow execution models

The first execution model is called the simple queue machine (SQM). This model uses a first-in, first-out queue for the manipulation of operands and results. In section II we describe this execution model and illustrate how it operates with an example. Next, in section III, we consider instruction sequences for the evaluation of arbitrary expressions. The instruction sequence required for the evaluation of an expression on an SQM can be easily and naturally derived from the parse tree for that expression. We specify the SQM model and present an algorithm for the generation of SQM instruction sequences from expression parse trees.

Copyright (c) 1987 by Bruno R. Preiss. All rights reserved.

Section IV shows that the SQM execution model has the potential for efficiently exploiting pipelined arithmetic/logic units (ALUs). The SQM model also supports a model of concurrent expression evaluation that is related to the evaluation of dataflow graphs. We show the performance potential of the SQM model by comparing it with stack-based execution.

The second execution model is called the indexed queue machine (IQM). It is treated in section V. This model uses a queue data structure that supports insertion of items anywhere in the queue. However, items are only removed from the front of the queue. We describe this execution model and illustrate how it operates with an example. Next, in section VI, we consider instruction sequences for computations expressed by arbitrary, acyclic dataflow graphs. We specify the IQM model and show that acyclic dataflow graphs can be considered as natural generators of IQM instruction sequences.

In section VII we address some of the issues in the practical implementation of the IQM execution model. In particular, we show that it is possible to implement a queue management scheme that supports the efficient execution of IQM instruction sequences. The basic technique is to use registers to act as a cache for the front of the operand queue.

Copyright (c) 1987 by Bruno R. Preiss. All rights reserved.

II. Simple Queue Machine Model

The SQM execution model uses a first-in, first-out (FIFO) queue data structure as the underlying control mechanism for the manipulation of operands and results. An SQM is analogous to a stack machine in that it has operations in its instruction set which implicitly reference an operand queue, just as a stack machine has operations which implicitly reference an operand stack[7]. In a stack machine, operands are retrieved (popped) from the top of an operand stack and results are returned (pushed) back onto the top of the operand stack. In a queue machine, operands are retrieved from the front of the queue of operands and results are returned to the rear of the queue of operands.

For example, consider the ADD instruction. On a stack machine, the ADD instruction pops two operands from the top of the operand stack, computes their sum, and pushes the sum back onto the stack. On an SQM, the ADD instruction removes two operands from the front of the operand queue, computes their sum, and places the sum at the rear of the queue. In the former case, a subsequent instruction cannot proceed until the result of a preceding instruction is returned to the top of the stack. In the latter case, the result is placed behind any other operands already in the queue. Consequently, a subsequent instruction may be started before the preceding instruction has completed, provided that its operands are available at the front of the queue. Thus, SQMs have the potential for efficiently exploiting multiple pipelined arithmetic/logic units (ALUs).

Copyright (c) 1987 by Bruno R. Preiss. All rights reserved.

A. A Sample SQM Instruction Sequence

As an example to compare the SQM execution model with the usual stack machine model, the instruction sequences required to compute the statement $f \leftarrow a \times b + (c - d)/e$ are shown in Table I. The table also shows the contents of the operand stack and operand queue after the instruction has completed execution. In the case of the stack machine, the top of stack is on the left. In the case of the SQM, the front of the operand queue is on the left. An important difference between the two execution models is the order of operands for dyadic operators. In the stack machine, the left operand of a dyadic operator is pushed first, followed by the right operand. Similarly, on an SQM the left operand of a dyadic operator is enqueued first, followed by the right operand. Consequently, in a stack machine the top of stack contains the right operand, whereas in an SQM, the front of queue contains the left operand.

In this example, the stack machine and SQM instruction sequences have the same length and use the same set of instructions. The SQM instruction sequence is simply a permutation of the stack machine instruction sequence. We will show that this is true in general due to the manner in which instruction sequences are generated for such machines.

It is well known that a stack machine instruction sequence to evaluate a given arithmetic expression can be obtained by doing a *post-order* traversal of the parse tree representing that expression[8]. Analogously, an SQM instruction sequence to evaluate a given arithmetic expression can be obtained by doing a different traversal of the parse tree called a *level-order* traversal. The example from Table I will serve to illustrate how a level-order traversal proceeds. These concepts will be presented formally in the following section.

Copyright (c) 1987 by Bruno R. Preiss. All rights reserved.

The parse tree corresponding to the statement of Table I is shown in Fig. 1 (a). The leaf nodes of the parse tree correspond to fetch operations and the internal nodes correspond to arithmetic operators. A level-order traversal of the parse tree of Fig. 1 (a) is obtained by visiting the nodes of the tree in the order shown in Fig. 1 (b). Informally, a level-order traversal results from visiting the nodes of the parse tree from the deepest to the shallowest levels and from left to right within each level. (The left-to-right direction is a consequence of the semantics of non-commutative, dyadic operations such as subtraction and division.)

Unfortunately, a level-order traversal done directly on the parse tree is inefficient and complicated because it involves back-tracking. However, a level-order traversal can be accomplished in two steps: First, a new tree called the *level-order conjugate* is constructed. Second, an *in-order* traversal of the level-order conjugate tree is performed. The level-order conjugate tree has the property that an in-order traversal of its nodes produces the desired instruction sequence. For example, the level-order conjugate of the tree of Fig. 1 (a) is shown in Fig. 1 (c).

III. Evaluating Expressions on a Simple Queue Machine

In this section we will show that:

1) An SQM can be used to evaluate an arbitrary arithmetic expression.

2) An SQM instruction sequence for an arbitrary arithmetic expression can be derived efficiently from the parse tree for that expression.

A. Basic Definitions and Notation

A binary tree T is either the empty tree or an ordered triple consisting of a root node, a left binary subtree, and a right binary subtree. Consider an arbitrary binary tree T. Let $nodes(T)$

Copyright (c) 1987 by Bruno R. Preiss. All rights reserved.

represents the set of nodes of the binary tree T. The set of nodes of an empty tree is empty. The set of nodes of a non-empty tree is the root of the tree together with the set of nodes of the left and right binary subtrees.

Consider an arbitrary binary tree T. Let $level(T,a)$ represent the level of the node a in the tree T, where $a \in nodes(T)$. The level of a node in a tree is its distance from the root. I.e., the level of the root of a tree is zero, the level of its descendents is one, etc.

B. Level-Order Precedence and Level-Order Traversal

In this section we introduce a relation on the set of nodes of a binary tree called level-order precedence.

Definition: Level-order precedence is a relation defined on the set of nodes of a binary tree, $nodes(T)$, such that for all pairs of nodes, a and b, the relation a precedes b (written $a <_T b$) is true if, and only if, either

i) $a=b$, or

ii) $a \neq b$ and $level(T,a) > level(T,b)$, or

iii) $a \neq b$ and $level(T,a) = level(T,b)$ and there exists some other node c in the tree such that a is in the left subtree of c and b is in the right subtree of c.

The level-order precedence relation, $<_T$, is a total order. Consequently, it is possible to construct a unique sequence of the nodes of a binary tree ordered according to their precedence. We call this sequence the *level-order traversal* of a binary tree.

C. Expression Parse Trees and Evaluation

Copyright (c) 1987 by Bruno R. Preiss. All rights reserved.

In the preceding section we presented the notions of level-order precedence and level-order traversal of arbitrary binary trees. In this section we consider binary trees as parse trees for expressions. A (binary) expression parse tree is a tree whose nodes represent nullary, unary, or binary operators. The number of immediate descendents of a node is equal to its arity.

We now define what it means to evaluate a sequence of operators on an SQM. Using this definition, it can be shown that evaluating the level-order traversal of a binary expression parse tree on an SQM correctly computes the expression[9].

Definition: The evaluation $evaluation(I)$ of a sequence of operators $I=(op_1, \cdots, op_k)$ on a queue machine is the sequence of states $(state_1, \cdots, state_f)$ in which each state $state_i=(I_i,Q_i)$ is an ordered pair consisting of $I_i=(op_i, \cdots, op_k)$ which represents that portion of the operator sequence yet to be evaluated and $Q_i=(operand_1, \cdots, operand_l)$ which represents the contents of the queue. The initial state is $state_1=(I,())$.

If the arity of the next unevaluated operator in $state_i$ is n, then the $state_{i+1}$ is obtained by

i) removing n operands from the front of the queue,

ii) removing the first unevaluated operator from the operator sequence, and

iii) appending it to the tail of the queue.

Note that this definition of the evaluation of a sequence of operators does not involve any computation. This definition of evaluation allows for the formal proof of correctness. Instead of placing the operator in the queue, an actual implementation would place the result of applying the operator to the operands in the queue.

It can be shown that in every state in the evaluation of level-order traversal of an expression parse tree, the operands available at the front of the operand queue for the next operator to be

Copyright (c) 1987 by Bruno R. Preiss. All rights reserved.

executed are the results of the descendent(s) of that operator. That is, if the next unevaluated operator is a unary operator, then the front of the operand queue contains is descendent in the expression parse tree. Similarly, if the next unevaluated operator is a binary operator, then the front of the operand queue contains is left descendent in the expression parse tree followed by its right descendent. Consequently, when the instruction sequence has terminated, the operand queue contains one operator which is the root of the expression parse tree.

D. Example

To illustrate the concepts developed above, we consider the statement $f \leftarrow a \times b + (c - d)/e$. The binary expression parse tree corresponding to this statement is shown in Fig. 1 (a). The set of nodes in the tree is

$nodes(T) = \{+, \times, a, b, \div, -, c, d, e\}$.

The level-order traversal of this binary expression parse tree is

$level\ order\ traversal(T) = (c, d, a, b, -, e, \times, \div, +)$.

This sequence is an instruction sequence for an SQM. The evaluation of this instruction sequence on an SQM is the sequence of states tabulated in Table II. Note that at each state in the evaluation, the front of the operand queue contains the correct operands for the next operation and that in the final state of the evaluation there is a single value in the operand queue which is the desired result.

E. Level-Order Conjugate

The level-order conjugate of a binary tree, *level order conjugate*(T), is a binary tree containing the same set of nodes as T, but the nodes in the conjugate tree are arranged so that the following conditions are true:

Copyright (c) 1987 by Bruno R. Preiss. All rights reserved.

1. For every node b in the right subtree rooted at node a in *level order conjugate*(T), nodes a and b are at the same level in T, (i.e., $level(T,a)=level(T,b)$), and a precedes b in level order (i.e., $a <_T b$).

2. For every node b in the left subtree rooted at node a in *level order conjugate*(T), node a is closer to the root of T than b, (i.e., $level(T,a)<level(T,b)$).

If the in-order traversal of a binary tree, *in order traversal*(T), is defined as the sequence of nodes obtained by appending in order the in-order traversal of the left subtree of T, followed by the root of T, followed by the in-order traversal of the right subtree of T, then it can be shown that an in-order traversal of the level-order conjugate tree of a binary tree is equivalent to a level-order traversal of the original tree. I.e.,

$$\textit{in order traversal}(\textit{level order conjugate}(T))=\textit{level order traversal}(T).$$

F. Level-Order Traversal Algorithm

An algorithm to perform a level-order traversal of a binary tree is shown in Fig. 2. This algorithm is easily generalized to trees having nodes with arbitrary fan-out. The level-order conjugate tree is constructed while doing a *reverse post-order* traversal (i.e., visit the root, traverse the right subtree, traverse the left subtree) of the expression parse tree. Since this algorithm does a recursive, reverse post-order traversal of the parse tree, its time complexity is $O(n)$, where n is the number of nodes in the tree, and since it constructs the level-order conjugate tree as it goes, it requires $O(n)$ space.

IV. PIPELINED EXECUTION OF SQM INSTRUCTION SEQUENCES

Copyright (c) 1987 by Bruno R. Preiss. All rights reserved.

In this section we will show that the SQM execution mechanism implemented with a pipelined arithmetic/logic unit (ALU) is more efficient than purely stack based execution. Fig. 3 shows the flow of data between the ALU, the memory, and the queue or stack. Observe that the SQM execution model has the potential to take advantage of a pipelined ALU when the queue of operands is not empty. On the other hand, pure stack based execution cannot exploit a pipelined ALU, because the results of one operation must be returned to the top of the stack before they can become the operands of the next operation. For this reason, hardware and software techniques have been used to overcome the difficulties associated with pure stack based execution in pipelined processors. E.g., dynamic dataflow analysis in the hardware[10] and scoreboarding[11-13] are used to control the issuing of instructions and may permit the issuing of instructions out of order. Software techniques can also be used to change the instruction sequence to optimize pipeline utilization.

To quantify the benefit to performance of the SQM execution model over the pure stack execution model on a pipelined ALU, we have computed the average percent reduction in the number of cycles needed to evaluate a (binary) expression parse tree with a given number of nodes on an n-stage pipelined ALU, Two cases are considered:

Case 1 (non-overlapped fetch/execute): The processor issues instructions at a rate of at most one instruction per cycle. Each instruction issued is either an ALU operation or an operand fetch. All ALU operations take exactly n cycles. An ALU operation cannot be issued until its operands are available at the top of the stack or the front of the queue. An operand fetch takes one cycle. To ensure that operands are pushed onto the stack in the correct order, an operand fetch cannot be issued until the ALU is idle.

Copyright (c) 1987 by Bruno R. Preiss. All rights reserved.

Case 2 (overlapped fetch/execute): The processor issues instructions at a rate of at most one instruction per cycle. Each instruction issued is either an ALU operation or an operand fetch. All ALU operations take exactly n cycles. An ALU operation cannot be issued until its operands are available at the top of the stack or the front of the queue. A fetch operation can be issued immediately and takes one cycle. Note, as a result of allowing fetches to be issued immediately operands are not necessarily pushed onto the stack in the correct order. An appropriate score-boarding scheme must be used to control the issuing of ALU operations. (E.g., see [11-13]).

Rather than comparing the execution times of arbitrary benchmark programs, we have computed the average execution time required to evaluate (binary) expression parse trees as a function of the number of nodes in the parse tree. Thus, the number of nodes in the parse tree is a measure of the complexity of the expression being evaluated. The computation of the average was done by exhaustively enumerating all possible (binary) expression parse trees with a given number of nodes (using the technique of [14]) and determining the number of cycles required to evaluate the expression on both SQM and stack machines with n-stage pipelined ALUs.

The results of the calculations are summarized in Tables III and IV. Note that the SQM execution model always meets or exceeds the performance of the stack machine. This is true for all instruction sequences (not just the average).

These tables also show the interesting result that under case 1, the benefit of the SQM over the stack machine increases with longer ALU pipelines. On the other hand under case 2, the benefit of the SQM is maximum for a two-stage pipelined ALU. This decreasing benefit is the result of the unrealistic nature of case 2 with respect to stack machines. By allowing the immediate issuing of fetch operations, the stack machine is allowed to do its pushes and pops out of order, which violates the basic stack execution mechanism.

Copyright (c) 1987 by Bruno R. Preiss. All rights reserved.

A. Concurrent Expression Evaluation

The SQM execution model can be used, without modification to support the concurrent evaluation of independent expressions. By evaluating expressions concurrently, it is possible to utilize a pipelined ALU more efficiently.

We will now describe the procedure for constructing instruction sequences for the concurrent evaluation of two, independent expressions. This procedure can be easily generalized to an arbitrary number of independent expressions. To construct an SQM instruction sequence to evaluate a pair of independent expressions, the expression parse trees for the expressions are joined together into a composite expression parse tree. The root node of the composite tree has no operation associated with it. Its immediate descendents are the root nodes of the expression parse trees. The SQM instruction sequence for the composite tree is generated exactly as for a simple parse tree (no operation is generated for the root of the composite tree).

When the instruction sequence for the composite tree is evaluated on an SQM, execution terminates with exactly two operands in the queue. These operands are the results of the evaluation of the component expressions.

In Table V, we show the percent change in the number of cycles required to evaluate an arbitrary pair of (binary) expression parse trees having the given number of nodes first consecutively and then concurrently on an SQM. (We use the case 1 assumptions described above.) For example, for a two-stage pipelined ALU, concurrent evaluation of instructions reduces the number of cycles required by up to about 25%. Note that as the number of nodes in the expression parse trees is increased, the advantage of concurrent expression evaluation decreases. This is because as the number of nodes increases, the SQM execution model utilizes the pipelined ALU more efficiently due to the parallelism inherent in the expressions themselves. This

Copyright (c) 1987 by Bruno R. Preiss. All rights reserved.

reduces the advantages of concurrent evaluation.

V. Indexed Queue Machine Model

In the preceding section, it was shown that stack machines and SQMs are equally powerful with respect to the evaluation of arbitrary expressions. However, both models also exhibit the same deficiency with regards to the evaluation of sequences of expressions (or simple assignment statements) where data dependencies are involved. This deficiency is the requirement for the explicit allocation of temporary memory locations or general-purpose registers to hold partial and temporary results that are needed to handle the data dependencies correctly.

The indexed queue machine (IQM) model of execution is an extension of the SQM model. This extension is motivated by the desire to overcome the deficiencies of stack machines and SQMs described above by absorbing the requirement for a temporary storage mechanism into a more complex queue insertion mechanism.

An IQM uses an indexed queue as the underlying control mechanism for the manipulation of operands and results[9, 15]. The result of each operation is assigned an index (or a set of indices). Each index specifies an offset from the front of the operand queue, which is assumed to occupy contiguous memory locations. The result is placed in the operand queue at the position(s) specified by the indices in the index set. Consequently, the result of an operation can be placed at any position(s) in the operand queue. However, operands are still only retrieved from the front of the operand queue. Thus, although results are inserted at arbitrary positions in the queue, operands are still always retrieved (implicitly) from the front of the queue, removing the requirement for explicit operand addressing and reducing instruction code space requirements.

Copyright (c) 1987 by Bruno R. Preiss. All rights reserved.

When constructing instruction sequences for an indexed queue machine, special care must be taken to ensure that the result of one operation does not overwrite the result of some other operation. Furthermore, the front of the queue must always contain valid operands — there must not be a hole in the queue.

A. A Sample IQM Instruction Sequence

As an example to illustrate the IQM execution model, the instruction sequence required to evaluate the dependent pair of statements $t \leftarrow a+b$; $e \leftarrow t \times c - t/d$ on an IQM is shown in Table VI. Note that the result index is specified as an offset from the front of the operand queue after the operands for the current instruction have been removed.

The sequence of assignments in the preceding example correspond to the acyclic dataflow graph (DFG) shown in Fig. 4. This DFG cannot be directly evaluated on an SQM or stack machine without using a temporary location for the value t. However, the IQM does not require the use of temporary locations. IQMs differ from SQMs and stack machines in their ability to be used to evaluate acyclic DFGs containing fan-out. Stack machines and SQMs are restricted to the evaluation of fan-out-free acyclic DFGs (i.e., inverted trees). Thus, the presence of nodes in a DFG having greater than unity fan-out translates into the requirement for temporary locations in stack machines and SQMs. IQMs overcome the temporary storage requirement by allowing instructions with multiple result indices (fan-out) that store their results at several queue positions (temporary storage).

VI. Computing Acyclic Dataflow Graphs on an Indexed Queue Machine

Copyright (c) 1987 by Bruno R. Preiss. All rights reserved.

In this section we show that, just as expression parse trees can be used as generators of instruction sequences for SQMs, acyclic dataflow graphs can be used as generators of instruction sequences for the IQM execution model. A dataflow graph is a directed graph in which the nodes represent operators and the arcs represent the flow of operands between operators. An acyclic dataflow graph corresponds to a sequence of (dependent) assignment statements in a high-level language. Consequently, acyclic dataflow graphs can be used to account for the use of temporary variables in a computation.

A. Basic Definitions and Notation

A directed graph $G=(V,E)$ consists of a set of vertices V and a set E of ordered pairs of vertices called edges. A path p in a graph is a sequence of its vertices $(v_1, \cdots, v_l)$ such that $\{(v_1,v_2),(v_2,v_3), \cdots ,(v_{l-1},v_l)\} \subseteq E$. A cycle is a path of length one or more that starts and ends at the same vertex. A directed acyclic graph is a directed graph with no cycles.

An (acyclic, binary) dataflow graph $G=(V,E)$ is a directed acyclic graph with the following properties:

i) its vertices, V, represent nullary, unary, or binary operators, and

ii) no edges end on nullary operators, exactly one edge ends on each unary operator, and exactly two edges end on each binary operator,

iii) each edge (v,w) ending at the same operator is given a unique label, $label(v,s)$, from the set $0,1,2, \cdots ,arity(w)-1$.

B. Level-Order Precedence in a Dataflow Graph

Copyright (c) 1987 by Bruno R. Preiss. All rights reserved.

In this section we introduce a relation on the set of nodes in an acyclic, binary dataflow graph that is analogous to the level-order precedence relation on the nodes of a binary expression parse tree. The intent is the same in both cases — the relation is used to define instruction sequences.

Definition: Level-order precedence is a relation defined on the set of the vertices V of a (binary, acyclic) dataflow graph $G=(V,E)$ such that for all pairs of vertices, v and w, the relation v precedes w (written $v <_G w$) is true if, and only if, either

i) $v=w$, or

ii) $v \neq w$ and there exists a directed path in G from v to w.

The level-order precedence relation $<_G$ is a partial order. Note, the relation $<_G$ is not a total order since it is possible for there to be a pair of nodes, v and w, for which neither $v <_G w$ nor $w <_G v$ hold.

C. Instruction Sequences and Evaluation

In this section we define instruction sequences for IQMs and how they can be constructed from acyclic, binary dataflow graphs. Since the level-order precedence relation defined above for binary, acyclic dataflow graphs is only a partial order, an instruction sequence derived therefrom is not necessarily unique.

In this section we also define what it means to evaluate an instruction sequence on an IQM. It can be formally shown that evaluating an IQM instruction sequence on an IQM correctly performs the computation represented by the acyclic, binary dataflow graph.

Copyright (c) 1987 by Bruno R. Preiss. All rights reserved.

Definition: An instruction sequence for an indexed queue machine is a sequence $I=(inst_1, \cdots, inst_k)$ of pairs $inst_i=(op_i, index\ set_i)$, where op_i is an operator and $index\ set_i$ is a (possibly empty) set of non-negative integers representing the indices of the result.

Definition: The evaluation of an indexed queue machine instruction sequence is the sequence of states $E(I)=(state_1, \cdots, state_f)$ where $state_i=(I_i, Q_i, front_i)$. $I_i=(inst_i, \cdots, inst_k)$ represents the portion of the instruction sequence yet to be evaluated, $Q_i=(operand_1, \cdots, operand_l)$ represents the contents of the operand queue ($l=1+\sum_{i=1}^{k} arity(op_i)$), and $front_i$ is the index of the front of the operand queue.

The initial state is $state_1=(I,(\varepsilon,\varepsilon, \cdots, \varepsilon),1)$.

Given $state_i=(I_i, Q_i, front_i)$ where $I_i=(inst_i, \cdots, inst_k)$, $Q_i=(operand_1, \cdots, operand_l)$, and $inst_i=(op_i, \{index_{i,1}, \cdots, index_{i,m}\})$, the next state, $state_{i+1}$, is computed as follows:

i) $I_{i+1}=(inst_{i+1}, \cdots, inst_k)$,

ii) $front_{i+1}=front_i+arity(op_i)$,

iii) Q_{i+1} is obtained from Q_i by replacing $operand_j$ with op_i for all values of j in $front_{i+1}+index_{i,1}, front_{i+1}+index_{i,2}, \cdots, front_{i+1}+index_{i,m}$.

Definition: Given an (acyclic, binary) dataflow graph G, a valid indexed queue machine instruction sequence is a sequence constructed as follows:

1) Choose a sequence $(op_1, \cdots, op_{|V|})$ of the nodes $op_i \in V$ such that for all pairs of operands op_i and op_j where op_i appears before op_j in the sequence (i.e., $i<j$), op_j does not precede op_i (i.e., $\neg(op_j <_G op_i)$). Such a sequence can always be constructed since $<_G$ is a partial order.

Copyright (c) 1987 by Bruno R. Preiss. All rights reserved.

2) Compute an offset sequence $(offset_1, \cdots, offset_{|V|})$ such that $offset_i = 1 + \sum_{j=1}^{i-1} arity(op_j)$.

3) Form the result index sets $index\ set_1, \cdots, index\ set_{|V|}$ such that $index\ set_i = \{(offset_j - (offset_i + arity(op_i)) + label(op_i, op_j)$: for $all\ j\ such\ that\ (op_i, op_j) \in E\}$.

4) Finally, form the instruction sequence $(inst_1, \cdots, inst_{|V|})$ such that $inst_i = (op_i, index\ set_i)$.

It can be shown that in every state in the evaluation of a valid indexed queue machine instruction sequence constructed as above, the operands available at the front of the operand queue for the next operator to be executed are the results of the predecessors in the dataflow graph of that operator[9].

D. Example

To illustrate the concepts developed formally above, we consider the earlier example of the sequence of statements $t \leftarrow a+b$; $e \leftarrow t \times c - t/d$. The binary, acyclic dataflow graph corresponding to this sequence of statements is $G=(V,E)$, where

$V=\{a,b,c,d,+,\times,\div,-\}$,

and

$E=\{(a,+,0), (b,+,1), (c,\times,1), (d,\div,1), (+,\times,0), (+,\div,0), (\times,-,0), (\div,-,1)\}$.

The graphical representation of this dataflow graph is shown in Fig. 4. The level-order precedence relation defined on the set of nodes V of this dataflow graph is tabulated in Table VII. Since the level-order precedence relation of the set of nodes of a dataflow graph is only a partial order, the operator sequence derived from it is not unique. In this example, there are several sequences that satisfy the level-order precedence relation. One such operator sequence is

$(a,b,+,c,\times,d,\div,-)$.

Copyright (c) 1987 by Bruno R. Preiss. All rights reserved.

The offset sequence for this operator sequence is

(1,1,1,3,3,5,5,7).

The resulting IQM instruction sequence for this operator sequence is

$((a,\{0\}), (b,\{1\}), (+,\{0,2\}), (c,\{1\}), (\times,\{2\}), (d,\{1\}), (\div,\{1\}), (-,\{0\}))$.

The evaluation of this instruction sequence on an IQM is the sequence of states tabulated in Table VIII. In this table, the operand at the front of the operand queue has been underlined. Note that at each state in the evaluation, the front of the operand queue contains the correct operands for the next operation and that in the final state of the evaluation the queue contains the desired results.

VII. Implementing an Indexed Queue Machine

A. Virtual and Physical Queues

In this section we describe a practical approach for implementing the IQM execution model. The basis of the IQM model is the use of a (virtual) queue to handle operands. Each instruction removes the required number of operands from the front of the queue, performs some computation, and stores the results in the operand queue at the specified offsets from the front of the queue.

The operand queue is easily implemented by mapping the virtual queue into a single page of physical memory. This mapping can be accomplished by splitting the QP into two parts. The most significant part is the page number and the least significant part is the page offset. As operands are consumed from the front of the queue, the page number remains constant, whereas the page offset is incremented modulo the size of the page.

Copyright (c) 1987 by Bruno R. Preiss. All rights reserved.

The maximum length of the virtual queue is limited by the size of a memory page. The size of a memory page also limits the utilization of operand queue memory, where utilization is defined as the fraction of memory locations that contain live data (i.e., data that will be used later). An upper-bound on the average utilization can be obtained by dividing the average queue length by the size of a memory page. This quantity is an upper bound because, in general, the operand queue may have fewer live values than its length due to the presence of holes in the queue. Thus, the choice of page size is a trade-off between the maximum queue length and average memory utilization.

B. Window Registers

The memory based implementation of the queue machine execution model described above has the property that all the instructions executed by the machine are memory-to-memory instructions. That is, operands for the instruction are fetched from memory and results are returned to memory. However, current implementation technologies have the property that memory access operations execute several times more slowly that most ALU operations. For this reason, it is desirable to use registers for operands wherever possible.

We now present an approach that uses $n=2^k$ CPU registers, called window registers, to hold the first n elements of the operand queue. That is, the window registers are associated with the memory locations $[QP]$, $[QP]\oplus 1$, $\cdots$, $[QP]\oplus(n-1)$, where $\oplus$ denotes addition modulo the page size and $[QP]$ denotes the contents of the QP register. The relationship between the (virtual) queue, its state in a given page of physical memory, and the associated window registers is shown in Fig. 5.

Copyright (c) 1987 by Bruno R. Preiss. All rights reserved.

Each window register has a presence bit that indicates either that the operand is available in the register, or that it must be retrieved from memory. The operand fetch phase of instruction execution begins by examining the presence bit of the window register associated with the front of the operand queue. If the bit is set, indicating that the register contains a valid operand, the operand is simply retrieved from the register. If the bit is not set, a memory fetch operation must be done to retrieve the operand from memory. In both cases, once the operand has been obtained, the presence bit is cleared and the QP is incremented so that it points at the next operand.

The result store phase of instruction execution begins by determining whether the offset associated with the result is smaller than n, the size of the window register set. If it is, the result is stored in the appropriate window register and the associated presence bit is set. Otherwise, the result is stored in the appropriate memory location.

If the instruction sequence has the property that most of the result indices associated with the results of instructions are less than the size of a register window, then many instructions will automatically execute in a register-to-register mode. Thus, the IQM execution model provides a way to efficiently evaluate dataflow graphs in a register based implementation. Furthermore, the IQM execution model automatically solves the register allocation and recovery problems. However, note that in this scheme values stored in registers are read exactly once. This means that the maximum achievable register utilization efficiency is lower that that for a conventional, Von Neumann architecture. However, the ability of this scheme to use registers at all makes it more efficient than dataflow architectures which also need tag matching to locate operands.

Copyright (c) 1987 by Bruno R. Preiss. All rights reserved.

C. Context Switches

When a processor does a context switch, it must first save the state of the currently executing context. It does so by *rolling out* the window registers whose presence bits are set by storing the register contents into their associated memory locations. Note that the presence bits themselves are not stored in memory. When a context resumes executing, it begins with all the presence bits cleared. In this way, operands are automatically restored by the normal execution mechanism described above without the need for explicitly restoring the state of the window registers after a context switch. In fact, it is not possible (nor necessary) to restore the exact state since the presence bits are not saved on a context switch.

The size of the window register set is an architectural parameter. The larger the set, the more efficient the execution mechanism. This is because more instructions will be executed in the register-to-register mode. The limiting case occurs when the window register set size is equal to the size of a memory page. In this case, all instructions would execute in the register-to-register mode. However, as the register window becomes larger, context switch overhead increases since more queue entries must be rolled out into memory.

The register window set clearly acts as a cache for the operand queue. In this case the cache line size is equal to one memory word (i.e. operand). However, the register window set differs from conventional associative or partially associative caches[16] in three ways:

First, at every point in the execution, there is a unique, one-to-one mapping from memory locations to cache locations. This mapping changes as the page offset part of the QP register is incremented. Since the mapping is one-to-one, associative cache search is not required.

Copyright (c) 1987 by Bruno R. Preiss. All rights reserved.

Second, values are only written into memory on a context switch. This differs from both store-through caches and copy-back caches. When a value is written into a store-through cache, it is also written into memory. When a value is written into a copy-back cache, the corresponding cache line is tagged (dirty bit) to indicate that the value has been changed. When the tagged cache line is to be replaced, it must be written back to memory. In the window register scheme replacement never occurs. Consequently, the only time that values are written back to memory are on a context switch.

Finally, the presence bit mechanism assures that only live values are written into memory. That is, only cache entries that will subsequently be used are written back into memory. The presence bit differs from a dirty bit in that it indicates both that the line has been changed and that the new value will subsequently be used. Dead values, i.e., values that have no subsequent use are not written back to memory, thereby reducing memory traffic.

VIII. CONCLUSIONS

Two queue machine execution models have been formally presented. It was shown that instruction sequences for the simple queue machine (SQM) execution model can be easily derived from expression parse trees. The SQM model can exploit a pipelined ALU more efficiently than a conventional stack machine. Furthermore, the SQM model supports concurrent expression evaluation which also increases pipeline utilization efficiency.

The indexed queue machine (IQM) execution model was shown to be suitable for the evaluation of acyclic dataflow graphs. Since instruction sequences for an IQM are determined statically, it is possible to exploit the techniques of instruction look-ahead and overlapped execution of sequential instructions. A practical approach to the implementation of the IQM model based on the use of a virtual queue and window registers that supports efficient execution was

Copyright (c) 1987 by Bruno R. Preiss. All rights reserved.

also described.

Copyright (c) 1987 by Bruno R. Preiss. All rights reserved.

References

1. A. L. Davis and R. M. Keller, ‘‘Data Flow Program Graphs,’’ *Computer*, Vol. 15, No. 2, Feb. 1982. pp. 26-41,
2. D. D. Gajski, D. A. Padua, D. J. Kuck, and R. H. Kuhn, ‘‘A Second Opinion on Data Flow Machines and Languages,’’ *Computer*, Vol. 15, No. 2, Feb. 1982. pp. 168-172,
3. A. H. Veen, ‘‘Dataflow Machine Architecture,’’ *ACM Computing Surveys*, Vol. 18, No. 4, Dec. 1986. pp. 365-396,
4. S. R. Vegdahl, ‘‘A Survey of Proposed Architectures for the Execution of Functional Languages,’’ *IEEE Trans. Comput.*, Vol. C-33, No. 12, Dec. 1984. pp. 1050-1071,
5. V. P. Srini, ‘‘An Architectural Comparison of Dataflow Systems,’’ *Computer*, Vol. 19, No. 3, Mar. 1986. pp. 68-88,
6. P. C. Treleaven, D. R. Brownbridge, and R. P. Hopkins, ‘‘Data-Driven and Demand-Driven Computer Architecture,’’ *ACM Computing Surveys*, Vol. 14, No. 1, Mar. 1982. pp. 93-143,
7. R. P. Blake, ‘‘Exploring a Stack Architecture,’’ *Computer*, Vol. 10, No. 5, May 1977. pp. 30-38,
8. D. E. Knuth, *Fundamental Algorithms*, Reading, Massachusetts: Addison-Wesley, 1973.
9. B. R. Preiss, ‘‘Data Flow on a Queue Machine,’’ Ph.D. Thesis, Dept. of Elec. Eng., Univ. of Toronto, 1987.
10. R. M. Tomasulo, ‘‘An Efficient Algorithm for Exploiting Multiple Arithmetic Units,’’ *IBM Journal of Research and Development*, Vol. 7, No. 1, Jan. 1967. pp. 25-33,
11. J. E. Thornton, *Design of a Computer: The Control Data 6600*, Glenview, Illinois: Scott, Foresman, and Co., 1970.
12. C. Melear, ‘‘The Design of the 88000 RISC Family,’’ *IEEE Micro*, Vol. 9, No. 2, Apr. 1989. pp. 26-38,
13. L. Kohn, ‘‘Introducing the Intel i860 64-Bit Microprocessor,’’ *IEEE Micro*, Vol. 9, No. 4, Aug. 1989. pp. 15-30,
14. M. Solomon and R. A. Finkel, ‘‘A Note on Enumerating Binary Trees,’’ *Journal of the ACM*, Vol. 27, No. 1, Jan. 1980. pp. 3-5,
15. B. R. Preiss and V. C. Hamacher, ‘‘Data Flow on a Queue Machine,’’ *Proc. of the 12th Ann. Symp. Comput. Arch.*, June 1985. pp. 342-351,
16. A. J. Smith, ‘‘Cache Memories,’’ *ACM Computing Surveys*, Vol. 14, No. 3, Sept. 1982. pp. 473-530,

Copyright (c) 1987 by Bruno R. Preiss. All rights reserved.

TABLE I
SQM AND STACK MACHINE
INSTRUCTION SEQUENCES TO EVALUATE
$f \leftarrow a \times b + (c-d)/e$

Stack Machine		SQM	
instruction sequence	stack contents	instruction sequence	queue contents
fetch a	a	fetch c	c
fetch b	b, a	fetch d	c, d
mul	ab	fetch a	c, d, a
fetch c	c, ab	fetch b	c, d, a, b
fetch d	d, c, ab	sub	$a, b, c-d$
sub	$c-d, ab$	fetch e	$a, b, c-d, e$
fetch e	$e, c-d, ab$	mul	$c-d, e, ab$
div	$(c-d)/e, ab$	div	$ab, (c-d)/e$
add	$ab+(c-d)/e$	add	$ab+(c-d)/e$
store f		store f	

TABLE II
EVALUATION ON AN SQM

$state_1$=	$((a,b,\times,c,d,-,e,\div,+),$	$()\,)$
$state_2$=	$((b,\times,c,d,-,e,\div,+),$	$(a)\,)$
$state_3$=	$((\times,c,d,-,e,\div,+),$	$(a,b)\,)$
$state_4$=	$((c,d,-,e,\div,+),$	$(\times)\,)$
$state_5$=	$((d,-,e,\div,+),$	$(\times,c)\,)$
$state_6$=	$((-,e,\div,+),$	$(\times,c,d)\,)$
$state_7$=	$((e,\div,+),$	$(d,-)\,)$
$state_8$=	$((\div,+),$	$(d,-,e)\,)$
$state_9$=	$((+),$	$(e,\div)\,)$
$state_{10}$=	$(),$	$(+)\,)$

Copyright (c) 1987 by Bruno R. Preiss. All rights reserved.

TABLE III
AVERAGE INSTRUCTION CYCLE REDUCTION PERCENTAGES: CASE 1
(NON-OVERLAPPED FETCH AND EXECUTE)

$$100\times\frac{Stack\ Cycles - SQM\ Cycles}{Stack\ Cycles}$$

nodes in parse tree	number of trees	number of pipeline stages 1	2	3	4	5
1	1	0.0	0.0	0.0	0.0	0.0
2	1	0.0	0.0	0.0	0.0	0.0
3	2	0.0	0.0	0.0	0.0	0.0
4	4	0.0	0.0	0.0	0.0	0.0
5	9	0.0	1.3	1.9	2.3	2.5
6	20	0.0	2.4	3.2	3.7	4.0
7	45	0.0	3.3	4.3	4.9	5.3
8	101	0.0	4.0	5.2	5.9	6.4
9	227	0.0	4.7	6.0	6.8	7.3
10	510	0.0	5.3	6.7	7.5	8.1
11	1146	0.0	5.7	7.3	8.2	8.8

TABLE IV
AVERAGE INSTRUCTION CYCLE REDUCTION PERCENTAGES: CASE 2
(OVERLAPPED FETCH AND EXECUTE)

$$100\times\frac{Stack\ Cycles - SQM\ Cycles}{Stack\ Cycles}$$

nodes in parse tree	number of trees	number of pipeline stages 1	2	3	4	5
1	1	0.0	0.0	0.0	0.0	0.0
2	1	0.0	0.0	0.0	0.0	0.0
3	2	0.0	0.0	0.0	0.0	0.0
4	4	0.0	0.0	0.0	0.0	0.0
5	9	0.0	1.3	1.0	0.8	0.7
6	20	0.0	3.0	2.6	2.1	1.7
7	45	0.0	4.5	4.0	3.3	2.8
8	101	0.0	5.7	5.2	4.4	3.9
9	227	0.0	6.7	6.3	5.4	4.8
10	510	0.0	7.5	7.3	6.3	5.6
11	1146	0.0	8.3	8.1	7.1	6.4

Copyright (c) 1987 by Bruno R. Preiss. All rights reserved.

TABLE V
AVERAGE INSTRUCTION CYCLE REDUCTION PERCENTAGES
CONCURRENT VS. CONSECUTIVE EXPRESSION EVALUATION

$$100\times\frac{Consecutive\ Cycles - Concurrent\ Cycles}{Consecutive\ Cycles}$$

nodes in parse tree	number of pipeline stages 1	2	3	4	5
1	0.0	0.0	0.0	0.0	0.0
2	0.0	25.0	30.0	33.3	35.7
3	0.0	25.0	28.6	30.9	32.5
4	0.0	24.6	27.6	29.5	30.8
5	0.0	23.9	27.2	29.1	30.4
6	0.0	23.1	26.9	28.8	30.1
7	0.0	22.4	26.6	28.6	29.9

TABLE VI
IQM INSTRUCTION SEQUENCE TO EVALUATE
$t \leftarrow a+b$; $e \leftarrow t\times c - t/d$

instruction sequence	result indices	queue
fetch a	0	a
fetch b	1	a, b
plus	0,2	t, ε, t
fetch c	1	t, c, t
mul	2	t, ε, $t\times c$
fetch d	1	t, d, $t\times c$
div	1	$t\times c$, t/d
sub	0	$t\times c + t/d$
store e		

TABLE VII
LEVEL-ORDER PRECEDENCE RELATION ON THE
NODES OF THE DATAFLOW GRAPH IN FIG. 4.

	a	b	c	d	$+$	$\times$	$\div$	$-$
a	$<_G$				$<_G$	$<_G$	$<_G$	$<_G$
b		$<_G$			$<_G$	$<_G$	$<_G$	$<_G$
c			$<_G$			$<_G$		$<_G$
d				$<_G$			$<_G$	$<_G$
$+$					$<_G$	$<_G$	$<_G$	$<_G$
$\times$						$<_G$		$<_G$
$\div$							$<_G$	$<_G$
$-$								$<_G$

Copyright (c) 1987 by Bruno R. Preiss. All rights reserved.

TABLE VIII
EVALUATION ON AN IQM

$state_1$=	$(((a,\{0\}),(b,\{1\}),(+,\{0,2\}),(c,\{1\}),(\times,\{2\}),(d,\{1\}),(\div,\{1\}),(-,\{0\})),$ $(\underline{\varepsilon},\varepsilon,\varepsilon,\varepsilon,\varepsilon,\varepsilon,\varepsilon,\varepsilon,\varepsilon),1)$
$state_2$=	$(((b,\{1\}),(+,\{0,2\}),(c,\{1\}),(\times,\{2\}),(d,\{1\}),(\div,\{1\}),(-,\{0\})),$ $(\underline{a},\varepsilon,\varepsilon,\varepsilon,\varepsilon,\varepsilon,\varepsilon,\varepsilon,\varepsilon),1)$
$state_3$=	$(((+,\{0,2\}),(c,\{1\}),(\times,\{2\}),(d,\{1\}),(\div,\{1\}),(-,\{0\})),$ $(\underline{a},b,\varepsilon,\varepsilon,\varepsilon,\varepsilon,\varepsilon,\varepsilon,\varepsilon),1)$
$state_4$=	$(((c,\{1\}),(\times,\{2\}),(d,\{1\}),(\div,\{1\}),(-,\{0\})),$ $(a,b,\underline{+},\varepsilon,+,\varepsilon,\varepsilon,\varepsilon,\varepsilon),3)$
$state_5$=	$((\times,\{2\}),(d,\{1\}),(\div,\{1\}),(-,\{0\})),$ $(a,b,\underline{+},c,+,\varepsilon,\varepsilon,\varepsilon,\varepsilon),3)$
$state_6$=	$((d,\{1\}),(\div,\{1\}),(-,\{0\})),$ $(a,b,+,c,\underline{+},\varepsilon,\times,\varepsilon,\varepsilon),5)$
$state_7$=	$((\div,\{1\}),(-,\{0\})),$ $(a,b,+,c,\underline{+},d,\times,\varepsilon,\varepsilon),5)$
$state_8$=	$((-,\{0\})),$ $(a,b,+,c,+,d,\underline{\times},\div,\varepsilon),7)$
$state_9$=	$(),$ $(a,b,+,c,+,d,\times,\div,\underline{-}),9)$

Copyright (c) 1987 by Bruno R. Preiss. All rights reserved.

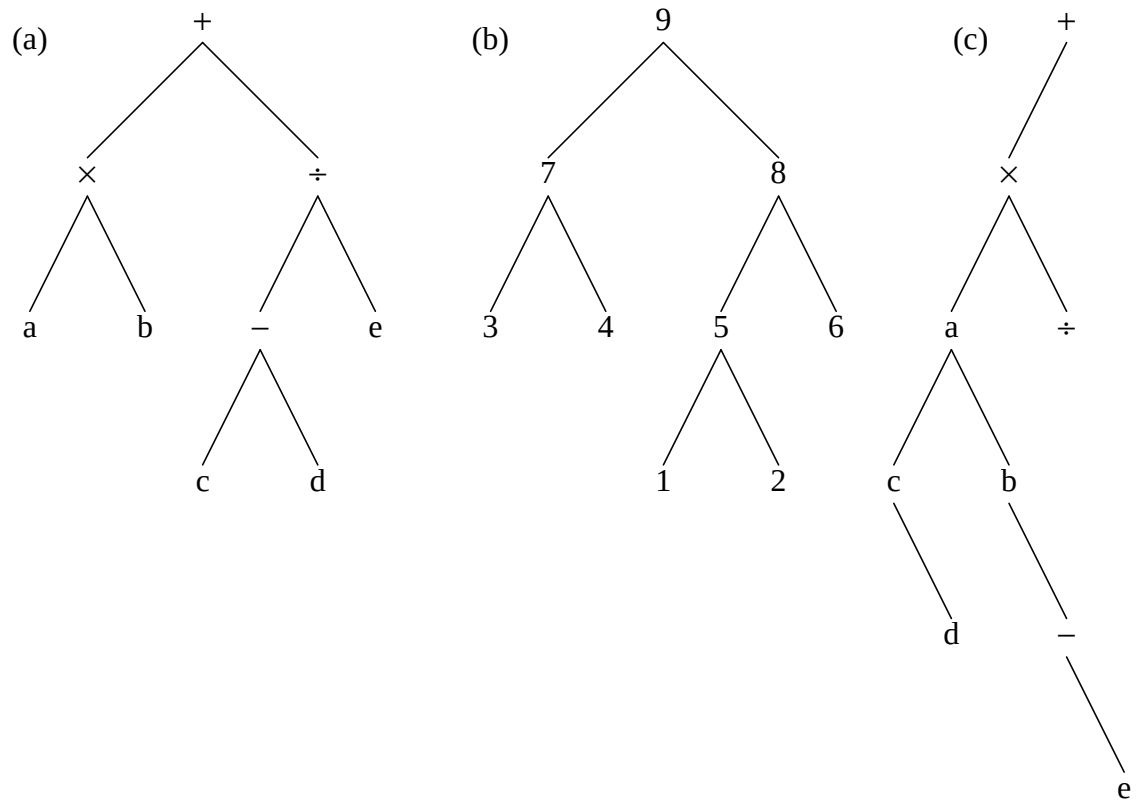

Fig. 1. The expression parse tree (a), level order of nodes (b), and level order conjugate tree (c) for the statement $f \leftarrow a \times b + (c - d)/e$.

Copyright (c) 1987 by Bruno R. Preiss. All rights reserved.

```
type
    nodePointer = ^ node;
    node =
        record
            left : nodePointer;
            right : nodePointer;
            contents : typeOfNodeContents
        end;

procedure LevelOrderTraverse (parseTree : nodePointer);

    var
        sentinel : nodePointer;

    procedure BuildConjugate (parseTree, conjugateTree : nodePointer);

        var
            tmp : nodePointer;
        begin
            if parseTree <> nil then begin
                new (tmp);
                tmp ^ .left := nil;
                if conjugateTree ^ .left = nil then begin
                    tmp ^ .right := nil;
                    tmp ^ .contents := parseTree ^ . contents;
                    conjugateTree ^ .left := tmp
                end else begin
                    tmp ^ .right := conjugateTree ^ .left ^ .right;
                    tmp ^ .contents := conjugateTree ^ .left ^ .contents;
                    conjugateTree ^ .left ^ .right := tmp;
                    conjugateTree ^ .left ^ .contents := parseTree ^ .contents
                end;
                BuildConjugate (parseTree ^ .right, conjugateTree ^ .left);
                BuildConjugate (parseTree ^ .left, conjugateTree ^ .left)
            end
        end;

    procedure InOrderTraverse (ptr : nodePointer);
        begin
            if ptr <> nil then begin
                InOrderTraverse (ptr ^ .left);
                { emit instruction for ptr ^ .contents }
                InOrderTraverse (ptr ^ .right);
                dispose (ptr)
            end
        end;

    begin
        new (sentinel);
        sentinel ^ .left := nil;
        sentinel ^ .right := nil;
        BuildConjugate (parseTree, sentinel);
        InOrderTraverse (sentinel ^ .left);
        dispose (sentinel)
    end;
```

Fig. 2. Level order traversal algorithm.

Copyright (c) 1987 by Bruno R. Preiss. All rights reserved.

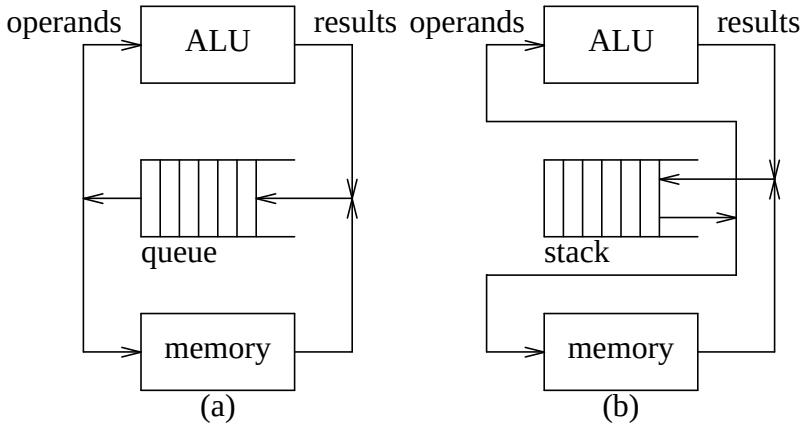

Fig. 3. Flow of operands and results between registers, ALU, and memory in the SQM model (a), and stack machine model (b).

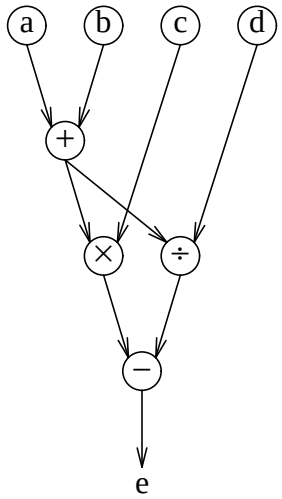

Fig. 4. Dataflow graph for the sequence of statements $t \leftarrow a+b$; $e \leftarrow t \times c - t/d$.

Copyright (c) 1987 by Bruno R. Preiss. All rights reserved.

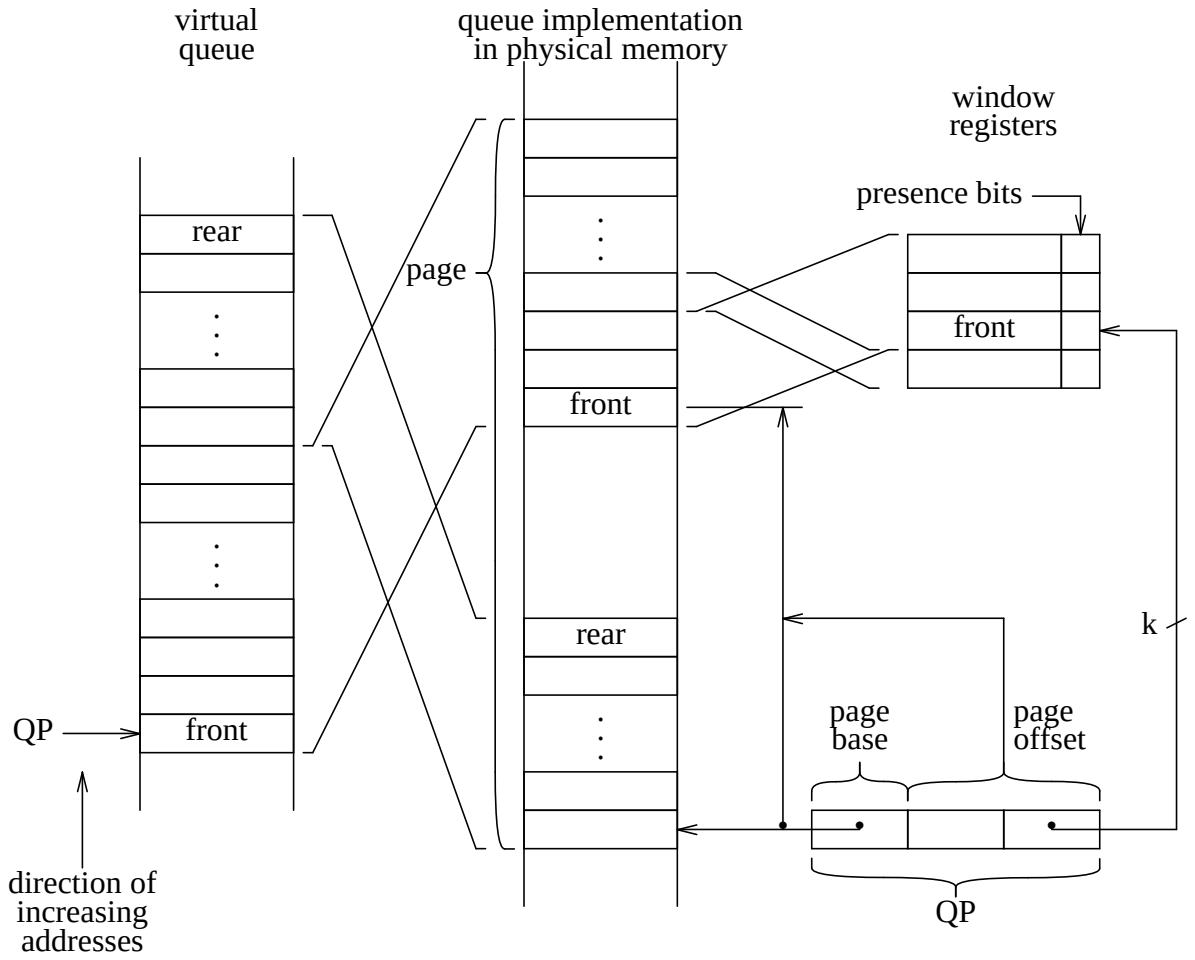

Fig. 5. Relationship between the virtual queue, the physical queue, and the window register set.

Copyright (c) 1987 by Bruno R. Preiss. All rights reserved.