A pipeline polish string computer*

by GERARD G. BAILLE and JEAN P. SCHOELLKOPF

Computer Architecture Group
Grenoble University, France

ABSTRACT

This paper describes a new computer organization which allows the pipeline execution of a polish-string code. The central characteristic of the proposed organization is a FI-FO queue, that holds values just accessed by an Access Station, until they are used as operands by an Execution Station for an arithmetic or logical operation. Operands are taken out of the queue by means of two pointers whose modifications are managed by a Control Station. This new computer organization is capable of high performance, since access to variables and execution of operations are performed in parallel with control functions required by the input program. A solution to access conflicts is proposed, using a content addressable memory that holds the names of the variables whose modification is deferred. This architecture is currently in application for the design of a high-level PASCAL computer.

INTRODUCTION

Design of high performance computers can be achieved using the technique that is called “pipeline” design, characterized by the fact that concurrent operations are supported by the machine. A high-level instruction can be initiated within a module, and in the same time the other modules are executing some operation related to a preceding instruction. The IBM 360/91 and the CDC 6600 are two examples of pipeline execution. Efficiency of pipeline computers depends strongly on the way that they are programmed. Many attempts were made to solve this problem and several techniques are proposed for generating optimized code in terms of pipeline execution. These techniques imply an important amount of preprocessing, which does not always justify the complexity of pipeline execution.

The solution proposed in this paper is based on a natural decomposition of the work to be executed in a pipeline manner. The first aspect of a natural decomposition is related to the kind of language proposed to the machine. Tomasulo gives an efficient algorithm for exploiting a pipeline architecture, but its application is limited to a low level language. On the other hand polish-string code appears as best suited for the execution of high level language.

Stones proposes a pipeline architecture for a push-down stack computer, but he suggests to translate the input polish-string code into three address instruction code. This paper shows, in a first section, how polish-string code can be directly executed by a pipeline computer. The second aspect of a natural decomposition, as shown by Abrams, is related to the decomposition of input string code execution into three natural processes that are control, access to operands and execution of operators. It would be interesting to have the three above processes concurrently running in a pipeline manner: the execution of an instruction would be initiated within the control station, and in the same time preceding instructions would be currently in process either within the access station or within the execution station. Such a pipeline organization is made possible using a FI-FO queue instead of a push-down stack as a work area for expression evaluation. Operands are accessed from the queue by means of two pointers whose modifications are controlled by the control station which generates extra-orders. The generation algorithm is presented and the rate of extra-orders is evaluated in the first section.

The second section of this paper gives the general architecture of the pipeline polish string computer. Concurrency between access to operands and evaluation of expression to be assigned to a variable leads to the well-known problem of reading the value of a variable whose modification is not yet performed. This problem, that is called “dependency” problem, is solved at execution time using a content-addressable memory which holds the names (not the address as in classical pipeline computers) of the variables whose modification is deferred.

A brief summary is given at the end of this paper, which shows the possible applications of the proposed pipeline architecture.

* The research reported on this project is sponsored by the French Comity for Computer Science Research under contract SESORI No. 74-156.
PIPELINE EXECUTION OF POLISH STRING CODE

This section first explains the choice of a FI-FO queue as a work area for evaluation of polish string. A model is introduced for polish string expressions which allows the definition of two pointers for operand access. The operator set is reduced to monadic and diadic operators. The management of the two pointers is controlled by special extra-orders generated by a Control station. The generation algorithm is presented and discussed.

The central characteristics of the pipeline execution is that the input polish stream is analyzed by a processor which generates two parallel instruction streams towards two concurrent processors specialized one for access to operands, the other for execution of operations.

Polish string evaluation

A polish string can be seen as a sequence of groups, each group being a sequence of operands followed by a sequence of operators. An operand is either an immediate value or the internal name of a variable (for example, lexical level and offset) which allows the calculation of the variable address in main memory.

example:

\[ [A,B,C,D,+,\ast,\ast] \quad [E,F,+,\ast,\ast] \quad [G,+,+,-] \]

group 1 \quad group 2 \quad group 3

The evaluation of such a Polish string requires the use of a working storage classically organized as a push-down stack.

All the operands of a given group are pushed, one after the other, into the stack, in the same order as in the input string, next operators are sequentially executed, popping the two topmost elements from the stack, and pushing the result onto the stack.

When all the operators of the current group are executed, the operands of the next group are pushed onto the stack, and the above process is performed again.

Therefore, two processes appear when evaluating a Polish string:

-an ACCESS process which stores the operands into the working storage (either from the string when immediate access, or from main memory in the other case)
-an EXECUTION process which executes the operators, taking its operands out of the working storage and storing intermediate results into it.

Using a push-down stack as working storage implies the sequentiality of the two above processes. The aim of this paper is to propose an evaluation method which allows parallel execution between ACCESS and EXECUTION process: parallelism is made possible using a FI-FO queue instead of a PUSH-DOWN stack.

Organization

The above two processes are executed by two independent processors. Let PAC be the processor which stores the operands into the FI-FO queue, and POP be the processor which executes the operators, taking its operands out of the queue.

Hence, when POP is executing an operator, PAC can Access to a new operand and store it into the queue.

example:

<table>
<thead>
<tr>
<th>Sequential execution (using a STACK)</th>
<th>Parallel execution (using a QUEUE)</th>
</tr>
</thead>
<tbody>
<tr>
<td>A B C D + E F + G + -</td>
<td>A B C D G F B P + G + - (PAC)</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>+ + * * + - (POP)</td>
</tr>
</tbody>
</table>

Such an organization requires a third processor which is called PAN, whose work is the ANalysis of the input polish string in order to generate instructions for both processors (PAC and POP).

The architecture is given by Figure 1.

THE EVALUATION PROCESS USING A FIFO QUEUE

Variables whose names appear in the input polish string are stored into the FIFO queue by the Access Processor. The sequence of the access instructions is the same as the sequence of the variable names in the input string. So, we can define the relative location of any variable in the queue, depending on its relative location in the input string.

Let \( S = (V_1', \ldots, V_i', O_1', \ldots O_{i''}, \ldots V_p', \ldots V_{p''}) \).
0_1 \ldots 0_\text{m} \) be the input string, where \( V_i \) is a variable name, and \( O_i \) is an operator name.

If the relative location of the first variable \( V_i \) is equal to 1, then the location of any variable \( V_j \) is given by the formula:

\[
1(V_i) = \sum_{r=0}^{i-1} n_r + j, \quad \text{with } n_0 = 0
\]

**How can we access to the operands located in the queue**

The evaluation of polish string is a sequence of monadic or diadic operators execution. In a first approach, let us consider that all the operators are diadic ones. We must associate to each operator its two operands, whose locations are partly defined by the next two rules:

**RULE 1**: if \( O_i \) is a diadic operator, then its second operand is variable \( V_{i+1} \), and its first operand is either variable \( V_i \) if \( n_i > 1 \), or the result of the immediately preceding operator \( O_{i-1} m_{i-1} \) if \( n_i = 1 \).

**RULE 2**: for \( j = 2 \) to \( n_i \), the second operand of operator \( O_j \) is the result of the preceding operator \( O_{j-1} \).

We see that the variables are not referred to as operands in the same order as in the input string:

for each group \( G_i = (V_i, \ldots, V_{i+n_i}, O_i, \ldots, O_{i+m_i}) \) the first accessed variables are \( V_{i+1} \) and \( V_{i+1+n_i} \) as operands for operator \( O_i \), the last accessed variable \( V_i \).

Hence, it is idle to pull the operands out of the queue in a FIFO mode, since it should be necessary to store them again into another memory, organized as a Pushdown stack.

So, we propose to use the queue as a working storage from which operands are accessed by means of two pointers, and into which intermediate results are stored during the evaluation process.

**Definition of two pointers**

Let us define \( P_1 \) and \( P_2 \) as two pointers which hold respectively the address of the first and second operand of any diadic operator during the evaluation process.

As an example, let \( S = (V_1, V_2, V_3, O_1, O_2, O_3) \) be the input string. The variables are stored into the queue in the following manner:

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>( V_1 )</td>
<td>( V_2 )</td>
<td>( V_3 )</td>
<td>( V_1 )</td>
<td>( V_2 )</td>
</tr>
</tbody>
</table>

\[ \uparrow \text{first-in} \quad \uparrow \text{last-in} \]

**Initialization of pointers**

Both pointers \( P_1 \) and \( P_2 \) must be initialized after execution of the last operator of group \( G_i = (O_i, m_i) \) and before execution of the first operator of the next group \( G_{i+1} = (O_{i+1}, m_{i+1}) \). We know that pointer \( P_2 \) gives the address of the location which holds the results of operator \( r(O_i, m_i) \) : this address is equal to \( 1(V_i+n_i) \). We must assign to \( P_2 \) the address of the second operand of operator \( O_i \), which is variable \( V_{i+1} \) (from rule 1), whose address is equal to

\[
1(V_i+n_i) = \sum_{r=0}^{i-1} n_r + n_i + n_i = 1(V_i+n_i) + n
\]

So we must increment the previous value of \( P_2 \) by \( n \). Moreover, pointer \( P_1 \) must hold the address of the first operand of operator \( O_i \), which is either variable \( V_{i+1} \) if \( n_i > 1 \), or the result of operator \( O_{i-1} m_{i-1} \) if \( n_i = 1 \).
The intermediate states of the queue

Each time a diadic operator is executed, the location which holds the first operand will be no more accessed: we say that a "hole" is created or the the location becomes "empty". So, during evaluation process, the state of the queue is defined as a sequence of empty locations followed by full locations which hold either not yet accessed variables or intermediate results. We associate to the state of the queue, after execution of any operator $O_i$ a finite sequence $\{ (d_1, t_1), (d_2, t_2), \ldots, (d_n, t_n) \}$, where each $d_i$ is the length of a full location sequence, and each $t_i$ is the length of an empty location sequence.

So, we define an EXTRA-ORDER, generated between the execution of $O_{ii}$ and the execution of $O_i$, called $UP(n_i)$, which is interpreted as:

$P_2$ must hold the address of the result $r(O_i)$, and $P_1$ must point to the full location downstream of the location pointed by $P_2$.

Each time an operator is executed, a new hole is created, therefore the last element of the sequence $\{ (d_i, t_i) \}$ must be modified: $t_i$ is incremented by 1 and $d_i$ is decremented by 1. During execution, pointer $P_1$ will be decremented by one. However, $d_i$ may become zero, in which case pointer $P_1$ must be decremented again by a value equal to $t_{i-1}$, in order to point to the first full location downstream of $P_2$.

In the same time, the queue state is updated, by suppressing the last element

\[ t_k + t_{k-1} + t_k : k \leq k-1 \]
Conclusion—An extra-order called DOWN must be generated by the PAN processor, when the queue state is updated; its execution is defined by: $P_1 \leftarrow P_1 - n$, where $n$ is equal to $t_k - 1$, when the order is generated.

Modification of the queue state before execution of a new group

Between execution of operator $O_{i-1}$ (the last operator of group $G_{i-1}$) and execution of operator $O_i$ (the first operator of group $G_i$), an extra-order UP must be generated. This movement of pointers corresponds to a modification of the queue state: a new element $(d_k, t_k)$ is created, initialized as follows: the new group $G_i$ is defined as a sequence of $n_i$ operands, hence $d_k \leftarrow n_i$, and zero hole has been created, hence $t_k \leftarrow 0$.

Conclusion—An extra-order called UP must be generated by PAN processor in order to update the queue state before the first operator of a new group. Its execution is defined by:

$$P_2 \leftarrow P_2 + n; P_1 \rightarrow P_2 - 1$$

where $n$ is equal to $d_k = n_i$, when the order is generated.

GENERATION OF EXTRA-ORDERS FOR THE MANAGEMENT OF THE POINTERS

The proposed evaluation process using a FIFO queue requires the generation of extra orders for the management of the pointers. As shown above, three kinds of orders must be sent to the execution processor:

1. The first kind consists in all the monadic or diadic operators, whose execution is defined by:

   $$Q(P_2) \leftarrow Q(P_1) <Op> Q(P_2)$$

   (diadic operator)

   or

   $$Q(P_2) \leftarrow <Op> Q(P_2)$$

   (monadic operator)

2. The arithmetic or logical diadic operation is followed by a modification of pointer $P_1$:

   $$P_1 \leftarrow P_1 - 1$$

3. The second kind is the UP (n) extra-order which is executed when a new group is entered. It is defined as:

   $$P_2 \leftarrow P_2 + n; P_1 \leftarrow P_2 - 1.$$ 

4. The last kind is the DOWN (n) extra-order, which consists in the updating of pointer $P_1$; it is defined as:

   $$P_1 \leftarrow P_1 - n$$

Evaluation of the number of extra-orders to be generated

The number of extra-orders ($n_{up} + n_{down}$) to be generated only depends on the input string.

Let $S = (V_1, \ldots, V_{p_1}, \ldots, V_{p_l}, \ldots, V_{p_M})$ be the input string.

Such a string contains

$$N = \sum_{i=1}^{p} n_i$$ variable names, hence $N$ access instructions for the access processor PAC, and

$$M = \sum_{i=1}^{p} m_i$$ operators, hence $M$ instructions for the execution processor POP. If all operators are diadic ones, then $M = N - 1$, but in general we have $M \geq N - 1$. Hence the initial number of orders is equal to $M + N \geq 2N - 1$

Each time a new group is entered, one must jump over its sequence of variables, that is to say that the number of extra UP orders to be generated is equal to the number $p$ of groups in the input string. The evaluation of the number of extra DOWN orders is a bit more difficult to do. Let $k_i$ be the number of extra DOWN orders generated during the execution of group $G_i$. The maximum value of $k_i$ is equal to $i - 1$, since the queue has a maximum number of holes equal to $i - 1$. Furthermore, each executed DOWN order decreases the number of further possible DOWN orders by 1, since one hole is suppressed.

Hence we have $0 \leq k_i \leq (i - 1) - \sum_{j=1}^{i-1} k_j$, for all $i$.

When adding the above formula for all $i$, we get:

$$0 \leq \sum_{j=1}^{p} k_j \leq \sum_{j=1}^{p} (j - 1) - \left( \sum_{j=1}^{p} k_j + \ldots + \sum_{j=1}^{p} k_p \right)$$

which becomes:

$$\sum_{j=1}^{p} (\sum k_j) \leq \sum_{j=1}^{p} (j - 1) = p(p - 1)/2.$$ 

Hence we have the number of extra Down orders given by

$$0 \leq \sum_{j=1}^{p} k_j \leq \frac{p(p - 1)}{2} - \frac{(p - 1)(p - 2)}{2} = p - 1.$$
However, if the number \( N \) of variables is less than \( 2p \) (\( N < 2p \)), than the maximum number of diadic operators is less than \( 2p - 1 \). As the \( p \) operators \( O_1, O_2, \ldots, O_p \) cannot imply the generation of a DOWN order, we have only \( p - 1 \) operators which can do so. Hence the maximum number of extra DOWN orders is equal either to

\[ N - p - 1 \] if \( p + 1 \leq N \leq 2p \), or to \( p - 1 \) if \( N \geq 2p \).

Now it is easy to see that the minimum rate of extra-orders is given by:

\[ \text{MIN RATE} \leq \frac{p}{2N-1} \]

and the maximum is given by:

\[ \text{MAX RATE} \leq \begin{cases} \frac{N-1}{2N-1} & \text{if } p + 1 \leq N \leq 2p, \\ \frac{2p-1}{2N-1} & \text{if } N \geq 2p \end{cases} \]

Example:

\( S = (A, B, C, *, +, D, -) \)

In this case, there are \( p = 2 \) groups, and \( N = 4 \) variables. Hence \( N \geq 2p \)

\[ \text{MIN RATE} \leq \frac{2}{7} \]
\[ \text{MAX RATE} \leq \frac{3}{7} \]

Trade off between compile time and execution time for the generation of the extra-orders

The generation of the input string is performed by the compiler, which could easily generate extra-orders. However, the compiler becomes machine dependent in such a case, and the size of generated code is increased, as the number of instruction fetches from main memory at execution time. Hence, the best solution consists in the analysis of the polish string at execution time, since the compile time solution would slow down the global performance of the machine. The generation is made by the analysis processor PAN, which can be considered as the control processor, and execute all the control function of the computer.

Algorithm for the generation of extra-orders at execution time

Generation of orders is performed by the analysis processor PAN. Given the input polish string, this processor must generate orders towards both processors PAC and POP, using the theoretic state of the queue, represented by the sequence \( \{(d_i, t_i)\} \) defined earlier. The sequence \( \{(d_i, t_i)\} \) can be managed using a push-down control stack. Let TS and STS be the two topmost elements of the stack, which respectively hold the couples \( (d_i, t_i) \) and \( (d_{i+1}, t_{i+1}) \). These two variables TS and STS are structured as two fields called \( D \) and \( T \), so \( d_i \) is equivalent to \( TS.D \), \( t_i \) to \( TS.T \) etc.

The generation algorithm is illustrated in Figure 9, where the symbol \( (F_i) \) represents the name of the function to be executed when the next symbol in the input string defines the state transitions. The next symbol type is represented either by \{variable\} or by \{operator\}.

![Figure 8](http://www.computerhistory.org)

![Figure 9](http://www.computerhistory.org)
Function F1 initializes the queue state, by pushing the couple \((-1,0)\) onto the control stack.

Function F2 generates an Access Order towards the Access Processor, next counts the number of variables in the current group, by incrementing the top of stack

\[(TS.D \leftarrow TS.D + 1)\]

Function F3 is executed when the first operator of the current group is encountered. It occurs on the transition from \(V_i^n\) to \(O_i^1\). Its function is the generation of an UP extra-order, with a parameter \(n\) equal to the number of variables in the current group, which has been evaluated by function F2 (the parameter is equal to \(n_i\) since F2 has been executed \(n_i\) times).

Function F4 generates an Operation order towards the execution processor POP. If the operator is a diadic one, then the queue state is updated, modifying the top of stack element STS: one generates DOWN (STS.T). Next the queue state is modified:

\[ST.S.D \leftarrow ST.S.D + TS.D \quad \text{TS.D} \rightarrow TS.D + 1\]

If TS.D becomes zero, an extra-order DOWN is generated, with a parameter \(n\) equal to the second top of stack element STS: one generates DOWN (STS.T). Next the queue state is modified:

\[ST.S.D \leftarrow ST.S.D + TS.D \quad \text{TS.D} \rightarrow TS.D + 1\]

The solution consists in the definition of a content addressable memory, organized in a FI-FO mode, which holds the name of the variables whose modification is deferred, and the name of the variables which have been just modified. In the first case (deferred modification), any reference to the variable is processed as an indirect reference by creating a link between the assignment and all the deferred references. In the second case a reference to main memory is eliminated, since the current value of the variable is available in the content-addressable memory after the completion of the last assignment.

Using the above mechanism, the access instruction \((\text{VALUE } X)\) can be deferred until the completion of the assignment. All the deferred references are linked together, eliminating a number of memory references equal to the number of linked locations.

### THE DEPENDENCY PROBLEM

Suppose, for example, that the high level instruction \(X \leftarrow \text{<exp>}\) has been prepared in both PAC and POP instruction queues, or is in the process of execution by processor POP, when the access to variable \(X\) instruction enters the access processor PAC. The access processor must be able to detect the fact that both instructions refer to the same variable \(X\), and that the second instruction might have to be deferred until the completion of the first instruction, since a reference to the location of \(X\) in main memory would not give the true value of variable \(X\), but its old value.

### A Pipeline Polish String Computer

The first section of this paper has shown that pipeline execution of polish string code is made possible using a FI-FO queue.

In this section, an architecture is presented for a high-level pipeline computer whose code is in polish string format. The dependency problem is first studied and a solution is given.

### THE CONDITIONAL BRANCH PROBLEM

Both PAC and POP processors may be considered as SLAVES of the PAN processor in the following sense: then only execute the internal instructions that they receive from the PAN processor. Moreover, every in-
struction belonging to the input string is fetched by the PAN processor. So, this processor can be considered as the MASTER of the control, and it is involved in all control functions in the computer during execution of a single high-level program.

When a conditional branch occurs, the PAN processor is not able to fetch the next instruction, since the conditional expression is currently in the process of evaluation. However, the PAN processor may choose one instruction among all the possible next instructions (generally two). The probability of a bad choice strongly depends on the context of the conditional branch: it is much lower for a LOOP statement than for an IF statement.

When a choice is made, we say that the PAN processor enters a Conditional State, characterized by the fact that its activity is limited to a preparation work. Especially, if a conditional branch occurs during this conditional state, no choice is made, the processor waiting for the resolution of the first conditional branch.

When the value of the conditional expression is available, two cases may occur: either the choice was good, in which case the process goes on without any modification, or the choice was bad, in which case all the prepared work must be disabled. This is simply achieved by writing as "empty" the input instruction queues of both PAC and POP processors which hold bad instructions and updating both evaluation and dependency queues by deleting the sequence of "bad" operands or "bad" deferred variables (they are "bad" because they belong to the bad choice).

HOW TO SAVE THE EVALUATION CONTEXT

The evaluation context (intermediate state of the evaluation queue) must be saved when a "function call" occurs within an expression. Function calls are introduced in the input polish code in the following manner: any operand can be either a variable name or a function call.

The syntax of a function call is

\[(\text{FCALL } <\text{name}> , <\text{parameter-list}> , \text{ENTER})\].

When FCALL is decoded by the analysis processor PAN, a special order is sent to the PAC instruction queue that calls for the address of a save area. This area is allocated on the top of a push-down stack, since several function calls can be nested. Next, processor PAN, which knows the current state of the evaluation queue, generates a sequence of DOWN and SAVE orders towards the POP instruction queue. Hence the current state of the evaluation queue can be saved before the function is entered, all previous results being compacted into the save area.

When the function is returned, processor PAC is capable to restore the initial state, pushing the function result just after the restored values, and the evaluation process goes on.

THE GLOBAL ARCHITECTURE OF THE COMPUTER

The pipeline computer architecture is shown by Figure 11. Each of the three processors can run concurrently. Their synchronization is data-driven. Processor PAN fetches high-level polish form instruction stream from Main Memory, and executes all the control instructions (IF, LOOP, GOTO, ...).

It analyzes input expressions and generates two internal instruction streams towards both PAC (access instructions) and POP (execution of operation).

Two synchronous exchanges occur: the first one is concerned with the transmission of the entry address when a procedure is entered. The second one is the completion of a conditional expression which can disable the choice made by processor PAN on the occurrence of a conditional branch.

SUMMARY

The pipeline architecture described in this paper is potentially capable of high performance. Its input code is in a polish string format that can be directly translated from a block structured high-level language. The pipeline execution is based on a natural decomposition into control, access to operands and execution of operations, executed by three concurrent processors.
This computer architecture is currently in application for the design of a PASCAL computer. Each of the three processors is in design using high speed macrologic components in Low Power Schottky Technology.11,12

ACKNOWLEDGMENTS

The authors would like to thank Dr. F. Anceau for his constant encouragement and helpful discussions and all members of the Computer Architecture Group of the University of Grenoble for their constructive criticism. Special thanks go to Mrs. H. Diaz for preparing the paper.

REFERENCES
