ISE-TR-83-34

# PERFORMANCE OF THE DIRECT-EXECUTION COMPUTER PASDEC

by

Kozo Itano

April 30, 1983

INSTITUTE
OF
INFORMATION SCIENCES AND ELECTRONICS

UNIVERSITY OF TSUKUBA

# Performance of the Direct-Execution Computer PASDEC

Kozo Itano
Institute of Information Sciences
and Electronics
University of Tsukuba
Sakura-mura, Niihari-gun, Ibaraki 305
Japan

This work was performed partly while the author was visiting at the Department of Computer Science, University of Maryland.

## Contents

Abstract

1. Introduction

2. Basic Concepts of PASDEC

   2.1 Tiny-Pascal Language
   2.2 Hardware Organization of the Computer
   2.3 Syntactical Sequences for the Direct-Execution

3. Principles of the Direct-Execution

   3.1 Declaration
   3.2 Subprogram Call
   3.3 If and While Constructs
   3.4 Assignment

4. Direct-Execution Performance

   4.1 Analysis of the Direct-Execution
   4.2 Improvement of the Design
   4.3 Example of the Direct-Execution Trace

5. Concluding Remarks

6. Acknowledgements

7. References

Fgires

Table

Abstract

A Pascal interactive direct-execution computer (PASDEC) for a subset of Pascal has been designed and simulated at the register transfer level using the Computer Design Language CDL [8]. In this paper, the direct-execution algorithm at the register transfer level is surmarized, and the execution performance of PASDEC is analyzed. The direct-execution time is given for a simple example, comparing it with the executions by the microprocessors.

# Performance of the Direct-Execution Computer PASDEC

Kozo Itano

## 1. Introduction

Despite of the rapid development of the hardware technology of a computer, programmer's productivity has not been improved drastically yet. One of the reasons for the low productivity of software lies in a poor environment caused by the semantic gaps between high-level languages and physical computers. Conceptually the direct-execution computer is known as an ideal computer which has no semantic gap between them. It can accept a high-level language program and directly execute it without compilation, assembly, likage editing or loading [1,2,4]. The direct-execution is particularly advantageous to an interactive programming environment [5], becuase a programmer can interact with his program execution naturally at the source code level during execution.

The motivation of the ongoing study of the direct-execution computer is to prove its usefulness in an interactive environment and feasibility of the hardware design. In this project, a Pascal Interactive Direct-Execution Computer (PASDEC) has been designed at the register transfer level using the Computer Design Language CDL [13-15]. The CDL simulation was also done to test the direct-execution algorithm of PASDEC and to evaluate its execution performance. This paper presents the analysis of the direct-execution and its performance evaluation based on the register transfer level design in CDL and its simulation results.

## 2. Basic Concepts of PASDEC

PASDEC is a direct-execution computer which executes Tiny-Pascal source program directly. Although Tiny-Pascal is a very simple program language, it still has block structure and data type which are typical features of a high-level program language.

### 2.1 Tiny-Pascal Language

Tiny-Pascal is a small suset of Pascal [9] which has the language constructs to perform simple programming. There are three control constructs: if-construct, while-construct, and subprogram; two data types: real and integer; two data structures: single variable and array; and one data flow operation: assignmnet.

### 2.2 Hardware Organization of the Computer

PASDEC consists of three major processors [8]: interactive processor, language processor, and I/O processor; and two memories: Program Memory (PM) and Data Memory (DM). The interactive processor executes interactive commands to enter a Tiny-Pascal source program and test it, and the language processor execute it directly. The I/O processor serves both these processors and executes I/O operations. The PM stores a Tiny-Pascal source program and the DM stores data value of program variables. As shown in Fig.1, the language processor consists of three subprocessors: Control Processor (CP), Data Processor (DP), and Lexical Processor (LP). The CP executes the control part of the program, and the DP executes the data part of it. The CP and DP activate the LP to fetch and assemble a token from the PM.

(1) Control Processor (CP)

The CP has two stacks: S-STACK and C-STACK; and one associative memory: Control Associative Memory (CAM). The S-STACK stores the information for the execution of the nested statements, and the C-STACK stores the return address and subprogram name. The CAM is seperated into two haves: CAM1 and CAM2. The CAM1 stores descriptors for subprograms, and the CAM2 stores descriptors for if-statement and while-statement.

(2) Data Processor (DP)

The DP has six stacks: ARP-STACK, N-STACK, P-STACK, V-STACK, A-STACK, and O-STACK; four registers: VAR-DESP, DM-OFFSET, NUM-PARM, and AR-SIZE; one table: VAR-DESP; and one associative memory: Data Associative Memory (DAM). The ARP-STACK stores activation record pointers. The P-STACK and N-STACK store actual parameters and number of them respectively on subprogram calls. The V-STACK stores value and type being evaluated. The O-STACK stores operators in the expression. VAR-DESP, VAR-LIST, and DM-OFFSET are used for the execution of subprogram declaration and subprogram calls.

(3) Lexical Processor (LP)

The LP has two registers: CHAR and CLASS; and one associative memory: Lexical Associative Memory (LAM). The LAM is seperated into two haves; the first half stores legal characters of the language and their corresponding classes; the other half stores reserved words of the language.

## 2.3 Syntactical Sequences for the Direct-Execution

As PASDEC has to recognize the program in a top-down manner to directly execute the program, a special syntax is formed so that each production rule has zero or one terinal symbols followed by zero or more non-terminal symbols. Based on the production rule, syntactical sequences are defined to control hardware components. As each production rule has one terminal symbol at most, the corresponding syntactical sequence recognizes one token at most. By this technique, the direct-execution can be easily interuptted or suspended after the execution of any token.

Using three registers: TOKEN, TYPE, and CODE; and one stack: CODE-STACK, the CP and DP determine the following syntactical sequences to be activated. For example, the first two production rules of the syntax is as follows:

<program> ::= PROGRAM <program-1>
<program-1> ::= <id> <program-2>

According to the production rule, the syntactical sequences 'program' and 'program-1' are defined. In the initial state, the register CODE indicates the syntactical sequence 'program' which expects to recognize a token 'PROGRAM'. When the proper token is fetched into register TOKEN, the next syntactical sequence 'program-1' is set to register CODE.

As the production rule may have more than one non-terminal symbol on its right side, the sequence corresponding to second or later non-terminal symbols should be kept on the CODE-STACK for the later execution.

There are four cases:

(1) Zero Non-Terminal

CODE-STACK is popped up and the top entry is set to register CODE as a sequence to be execute next.

(2) One Non-Terminal

A sequence corresponding to the non-terminal symbol is set to register CODE.

(3) Two Non-Terminals

A sequence corresponding to the first non-terminal symbol is set to register CODE, and a sequence corresponding to the second non-terminal is pushed on the CODE-STACK.

(4) Three or More Non-Terminals

A sequence corresponding to the first non-terminal symbol is set to register CODE, and sequences corresponding to the second and later non-terminals are pushed on the CODE-STACK in reversed order.

When the production rule has more than one alternative rules with no terminal symbol on its right side, some semantic action is needed to select one of them. The production rule for <statement> is a typical example:

```
<statement>   ::= IF  <if-1>
                | WHILE  <while-1>
                | BEGIN  <statement-list> <compound-end>
                | <proc-statement>
                | <asg-statement>
```

In this case, one of the sequences 'proc-statement' or 'asg-statement' should be selected when the token is a name. If it is a subprogram name, the sequence 'proc-statement' is the next syntactical sequence, and if it is a variable name the syntactical sequence 'asg-statement' is the next one. To determine this, associative memories should be referenced.

Sometimes, a token is recognized by more than one syntactical sequence. For example, a token ';' is first recognized as a seperator of an expression, then it is recognized as a sepaerator of a statement. This kind of situation will happen when the corresponding production rule includes <empty> on its right side.

# 3. Principles of the Direct-Execution

## 3.1 Declaration

When variables and subprograms are declared, corresponding descriptors are created and stored in the associative memories. A descriptor for a subprogram stores its type, subprogram name, location of the subprogram body, number of formal parameters, and size of the activation record. A descriptor for a variable stores its name, corresponding subprogram name, type, structure, size, location in the activation record, and a flag identifying whether it is a parameter.

When PASDEC encounters declarations, it executes them token-by-token. Suppose a simple subprogram decralation as shown below:

```
procedure swap(var a,b: integer);
var  x,y: integer;
begin
:
:
end;
```

The execution is performed in the following way. The data flow during the execution of declarations is given in Fig.2.

(1) A reserved word 'procedure' is recognized, and execution of subprogram declaration is initiated.

(2) The subprogram name 'swap' is recognized, then a new descriptor is created for this subprogram and stored in the CAM1. The name is also stored in register PROCID.

(3) As the subprogram has two formal parameters 'a' and 'b', they are recognized and stored in Table VAR-LIST.

(4) The data type 'integer' is recognized, and register VAR-DESP is set with corresponding value to 'integer'; four fields of the register: type, struc, size, and is-parm are set with 'integer', 'single-variable', '1', and 'yes' respectively.

(5) Descriptors for the parameters in table VAR-LIST are created and stored in the DAM one by one.

(6) Local variable 'x' and 'y' are recognized and stored in table VAR-LIST.

(7) Data type 'integer' is recognized, and register VAR-DESP is set properly; four field of the register are set with 'integer', 'single-variable', '1' and 'no'.

(8) Descriptors for local variables in VAR-LIST are created and stored in the DAM one by one.

(9) Subprogram body is found by detecting 'begin'; at this point, the number of formal parameters, size of the activation record (number of local variables), and location of subprogram body are stored in the CAM1.

(10) The subprogram body is skipped until the end of the body is found.

## 3.2 Subprogram Call

Two kind of subprograms: procedure and function are available in the Tiny-Pascal. When a subprogram is called, the computer refers a corresponding descriptor in the CAM1 to get necessary information for subprogram call. As the local variables are allocated as an activation record in the DM, the size of the activation record in the descriptor is sent to the DP on the subprogram calls. The number of formal parametrs in the descriptor is also sent to the DP to check the number of formal and actual parameters.

A simple example of subprogram call is given below:

```
begin
   :
   swap(i,j);
   :
end;
```

The subprogram 'swap' is the one defined in section 3.1. The execution is performed in the following way. The data flow during the execution is given in Fig.3.

(1) The name 'swap' is recognized as a subprogram name by refering the CAM1.

(2) The number of formal parameters and the size of the activation record in the descriptor are sent to the DP through registers: NUM-PARM and AR-SIZE.

(3) The DP recognizes the actual parameters 'i' and 'j'. It checks number of formal and actual parameters, refers the DAM to get their physical

addresses in the DM and types, then pushes them onto the P-STACK. Content of register NUM-PARM is pushed onto the N-STACK.

(4) Using AR-SIZE, a new activation record for the called subprogram 'swap' is allocated in the DM space. This is done by pushing the address of a new activation record on the ARP-STACK.

(5) The content of the program counter NEXT-PTR and subprogram name in register PROCID are saved in the C-STACK as a return address and previous subprogram name.

(6) A new address of the called subprogram body and subprogram name are stored in NEXT-PTR and PROCID.

(7) Subprogram body is executed.

(8) After executing the subprogram body, control is returned to the previous program. The ARP-STACK, N-STACK, P-STACK, and C-STACK are popped up, and content of registers: NUM-PARM, AR-SIZE, PROCID, and NEXT-PTR are recovered.

(9) If the subprogram is a function, the result'true' or 'false' is sent to the CP from the DP as a result of boolean expression at this point.

### 3.3 If and While Constructs

Execution of if and while statements may cause break of continuos control flow. The CP does not know where to jump at the first time it encounters these statements. Hence, it distinguishes the first time execution from the second time execution. At the first time execution, complete information of the control construct is established in the CAM2. At the second time execution, the CAM2 is refered and the next control flow is determined very fast.

Since the Tiny-Pascal permits nested statement of the control constructs such as if and while, execution of a control construct may not be finished until the execution of all the constructs inside of the current one are finished. In order to execute such nested constructs, the identification of the outer construct is pushed onto the S-STACK to continue the remaining execution later.

A simple example of the nested statements is given below:

```
while  a>0 do
   begin
      :
      while i<j do j:=j+1;
      :
   end;
```

The execution of this statement is performed in the following way. The data flow for the execution of the control constructs is shown in Fig.4.

(1) The first token 'while' is recognized, and the location in the PM is

pushed on the S-STACK; a new descriptor for this while construct is created and stored in the CAM-2.

(2) The location of the boolean expression is stored in the descriptor, and the expression is evaluated.

(3) When the token 'do' is encountered, the computer determines whether the following statement should be executed, according to the result of the boolean expression.

(4) If the while body is to be executed, execution continues.

(5) The second token 'while' is recognized, and the location in the PM is pushed on the S-STACK; a new descriptor for this is created and stored in the CAM-2.

(6) The boolean expression is recognized, and the location is stored in the current while descriptor.

(7) After reaching the end of the while body';', the location of the end of the current while construct is stored in the descriptor. At this point, a flag 'complete' is set to the descriptor for the inner while construct.

(8) S-STACK is poped up, and the outer while construct becomes the current construct again.

(9) After reaching the end of the outer while body ';', the location of the end of the outer while construct is stored in the descriptor in CAM-2. At this point, a flag 'complete' is set to the descriptor for the outer while construct.

## 3.4 Assignment

Assignment is only one data operation in the Tiny-Pascal language. When PASDEC encounters an assignmnet statement, the right side expression is evaluated and the result value is stored into the left side variable. Suppose a simple example of assignment statement as shown below:

```
a := b + 1;
```

The execution of this statement is performed in the following way. The data flow during the execution is shown in Fig.5.

(1) A variable name 'a' is recognized, and the execution of an assignmnet is initiated.

(2) The DAM is referenced to get the descriptor of 'a' which contains the type, structure, size, is-parm flag, and dm-offset. This information is pushed on the A-STACK. Before pushing it, the dm-offset is transformed into the physical location in the DM by adding the top entry of the ARP-STACK.

(3) The operator ':=' is recognized, and pushed on the O-STACK.

(4) The right side of ':=' is evaluated. At first, a variable name 'b' is recognized, and the corresponding descriptor is refered from the DAM. Then it is pushed on the A-STACK. Corresponding value is fetched from the DM and pushed on the V-STACK with the data type 'integer'.

(5) The operator '+' is recognized, and pushed on the O-STACK.

(6) A constant '1' is recognized, and pushed on the V-STACK.

(7) A terminator of an expression ';' is recognized. Hence, the expression on the V-STACK and O-STACK is evaluated.

(8) The result is assigned to the variable 'a'.

## 4. Direct-Execution Performance

The language processor of PASDEC has been designed at the register transfer level using the Computer Design Language (CDL) [13-15], and the design has been actually simulated by the CDL3 simulator on the UNIVAC-1180. Simulation has been done for several sample programs which include every control constructs, data operations, declarations, and data structures except for data type 'real'. Based on the simulation, the direct-execution performance was analyzed and some improvement was made.

### 4.1 Analysis of the Direct-Execution

To analyze the direct-execution performance, a simple program example is examined. The program consists of one while construct shown as:

```
while count>=0 do count:=count-1;
```

For the evaluation of the performance, it is compared with the performance of the equivalent programs for the microprocessors Z80 and Z8000. Detailed execution times and size of code are shown in Table 1.

The program for Z80 in Table 1-1 processes 8 bit data, and the program for Z8000 processes 16 bit data. In the CDL simulation, the data width of the Data Memory of PASDEC is preliminary 8 bit; however it is not strictly restricted. The microprocessors Z80 and Z8000 expect that memroy access time is 2-3 clock periods, although PASDEC expects that it is one clock period. Associative memories are expected to operate within one clock period too. The execution time of PASDEC in Table 1-3 is shown with 5 different categories of processing: lexical processing (LP), token recognition by Control Processor (CP), semantic

actions by Control Processor (CPS), token recognition by Data Processor (DP), semantic actions by Data Processor (DPS), and data operation by Data Processor (DPO). In this case, all these processings are not supposed to be overlapped. And each source character is fetched one by one from the PM which is a byte oriented memory.

Execution times are given in clock periods in a form of 'a + bn', where n indicates the number of iteration of the loop. Total execution times for Z80, Z8000, and PASDEC are 30+73n, 23+55n, and 68+125n respectively; hence, if the loop is iterated 10 times, they are 760, 573, and 1318 clock periods.

## 4.2 Improvement of the Design

For the speed up of the direct-execution, the direct-execution algorithm was improved. Improved pointes are enumerated below.

(1) A source program is assembled into tokens when it is entered in the PM.

(2) Lexical processing only manipulates string data, number, and comments.

(3) Lexical processing is completely overlapped with the later stage of the execution by CP and DP.

(4) Associative memory related operations are elaborated.

(5) Semantic actions are overlapped with the token recognition as far as possible.

(6) CAM1 and DAM are checked in parallel when a name is encountered as the first token of a statement.

Based on this improvement, the execution time was estimated as shown in Table 1-4. In the improved design, PASDEC can execute the same statement shown in section 4.1 within 43+71n clock periods. When it is iterated 10 times, the execution time is 753 clock periods.

### 4.3 Example of the Direct-Execution Trace

Direct-execution trace based on the simulation for the simple bubble sort program is shown in Fig.6. In this simulation the lexical processing was simplified to a simple fetching of the pre-assembled token to save the simulation time, and some semantic actions were overlapped with the recognition of the next tokens. This program contains 120 tokens, which was executed within 1809 clock periods. The UNIVAC 1180 CPU spent 549 seconds for this simulation. The first 156 clock periods were spent for executing the declarations, and the rest of them were spent for executing the program body. The mean direct-execution speed of this sorting program is approximately 8 clock periods per token.

## 5. Concluding Remarks

An in-depth hardware design of the direct-execution computer has been made for a simple subset of Pascal at the register transfer level using the Computer Design Language CDL. The CDL simulation of the computer was actually done on the UNIVAC-1180 at the University of Maryland. By the comparison of PASDEC with some microprocessors, we can conclude that the PASDEC executes the source program in very high speed. Concerning the size of code, a source program may be slightly bigger than microprocessor's binary code, although PASDEC has only one level source program. That it, PASDEC doesn't need to keep many level of codes such as complied relocatable bainary or load module, and the size of code is smaller than micorprocessor's one totally.
Based on the current design, the algorithm should be elaborated so that it becomes simpler and more understandable in addition to its speed. Associative memories should be analyzed in the detailed level for the actual implementation also.

## 6. Acknowledgements

The author wishes to acknowledge that his paper was originated partly from reference [3]. He also thanks to Professor Yaohan Chu for his helpful suggestions and discussions.

## 7. References

[1] Y. Chu, "High-Level Language Computer Architecture", Academic Press, 1975.

[2] Y. Chu, "A JOVIAL Direct-Execution Computer", Proc of High-Level Language Computer Architecture Conference, 1980, pp.17-32.

[3] K. Lor and Y. Chu, "Design of a Pascal Interactive Direct-Execution Computer", TR-1088, Department of Computer Science, University of Maryland, 1981.

[4] Y. Chu and M. Abrams, "Programming Languages and Direct-Execution Computer Architecture", Computer, Vol.14, No.4, July 1981, pp.22-32.

[5] Y. Chu, K. Itano, Y. Fukunaga, and M. Abrams, "Interactive Direct-Execution Programming and Testing", Proc of COMPSAC'82, Chicago, Nov. 1982.

[6] K. Itano and Y. Chu, "A Pascal Interactive Direct-Execution Computer: PASDEC, Part I: High-Level Design", TR-1198, Department of Computer Science, University of Maryland, August 1982.

[7] K. Itano, "A Pascal Interactive Direct-Execution Computer: PASDEC, Part II: CDL Design and Simulation", TR-1202, Department of Computer Science, University of Maryland, August 1982.

[8] K. Itano, "PASDEC: A Pascal Interactive Direct-Execution Computer", Proc. of High-Level Language Computer Architecture Conference, 1982.

[9] K. Itano, "CDL Design of a Pipelined Lexical Scanner", TR-1093, Department of Computer Science, University of Maryland, September 1981.

[10] A. Aho and J. Ullman, "Principles of Compiler Design", Addison-Wesley, 1979.

[11] Y. Chu, "Software Blueprints and Examples", Lexington Books, 1982.

[12] Y. Chu, "Computer System Design Description" Proc. of Design Automation Conference, Las Vegas, June 1982.

[13] Y. Chu, "An ALGOL-Like Computer Design Language", CACM, Vol.8, No.10, 1965, pp.607-615.

[14] Y. Chu, "User's Manual for the CDL3 Simulator", Department of Electrical Engineering, University of Maryland, May 1978.

[15] Y. Chu, "How to use the CDL3 Simulator on the UNIVAC 1100", Department of Electrical Engineering, University of Maryland, May 1978.

Table 1. Comparison of Execution Times

(1) Z80 Total: 19 bytes, 30+73n clock periods

```
                          bytes  clock
loop: ld  a,count           3     13
      cp  0                 2      7
      jp  m,exit            3     10
      ld  a,count           3     13
      sub 1                 2      7
      ld  count,a           3     13
      jp  loop              3     10
exit: ...
```

(2) Z8000 Total: 24 bytes, 23+55n clock periods

```
                          bytes  clock
loop: ld  a,count           4      9
      cp  a,0               2      7
      jp  mi,exit           4      7
      ld  a,count           4      9
      sub a,1               2      4
      ld  count,a           4      9
      jp  loop              4      7
exit: ...
```

(3) PASDEC (original) Total: 33 bytes, 68+125n clock periods

| | bytes | LP | CP | CPS | DP | DPS | DPO | total | CAM1 | CAM2 | DAM | DM |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| whileb̸ | 6 | 6 | 5 | 4 | 0 | 0 | 0 | 15 | | * | | |
| count | 5 | 5 | 0 | 0 | 9 | 8 | 0 | 22 | | * | * | * |
| >= | 2 | 2 | 0 | 0 | 3 | 0 | 0 | 5 | | | | |
| 0b̸ | 2 | 2 | 0 | 0 | 5 | 1 | 0 | 8 | | | | |
| dob̸ | 3 | 3 | 2/4 | 0/2 | 6 | 0 | 3 | 14/18 | | * | | |
| count | 5 | 5 | 3 | 3 | 4 | 5 | 0 | 20 | * | | * | |
| := | 2 | 2 | 0 | 0 | 2 | 0 | 0 | 4 | | | | |
| count | 5 | 5 | 0 | 0 | 9 | 8 | 0 | 22 | | | * | * |
| - | 1 | 1 | 0 | 0 | 3 | 0 | 0 | 4 | | | | |
| 1 | 1 | 1 | 0 | 0 | 5 | 1 | 0 | 7 | | | | |
| ; | 1 | 1 | 7 | 3 | 6 | 0 | 2 | 19 | | * | | * |

(4) PASDEC (improved) Total: 22 bytes, 43+71n clock periods

| | bytes | LP | CP | CPS | DP | DPS | DPO | total | CAM1 | CAM2 | DAM | DM |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| whileb̸ | 2 | (1) | 5 | 4 | 0 | 0 | 0 | 9 | | * | | |
| count | 2 | (1) | 0 | 0 | 5 | 5 | 0 | 10 | | * | * | * |
| >= | 2 | (1) | 0 | 0 | 3 | 0 | 0 | 3 | | | | |
| 0b̸ | 2 | (1) | 0 | 0 | 5 | 1 | 0 | 6 | | | | |
| dob̸ | 2 | (1) | 2/4 | 0/2 | 6 | 0 | 3 | 11/15 | | * | | |
| count | 2 | (1) | 3 | (3) | 4 | 5 | 0 | 12 | * | | * | |
| := | 2 | (1) | 0 | 0 | 2 | 0 | 0 | 2 | | | | |
| count | 2 | (1) | 0 | 0 | 5 | 5 | 0 | 10 | | | * | * |
| - | 2 | (1) | 0 | 0 | 3 | 0 | 0 | 3 | | | | |
| 1 | 2 | (1) | 0 | 0 | 5 | 1 | 0 | 6 | | | | |
| ; | 2 | (1) | 7 | 3 | 6 | 0 | 2 | 18 | | * | | * |

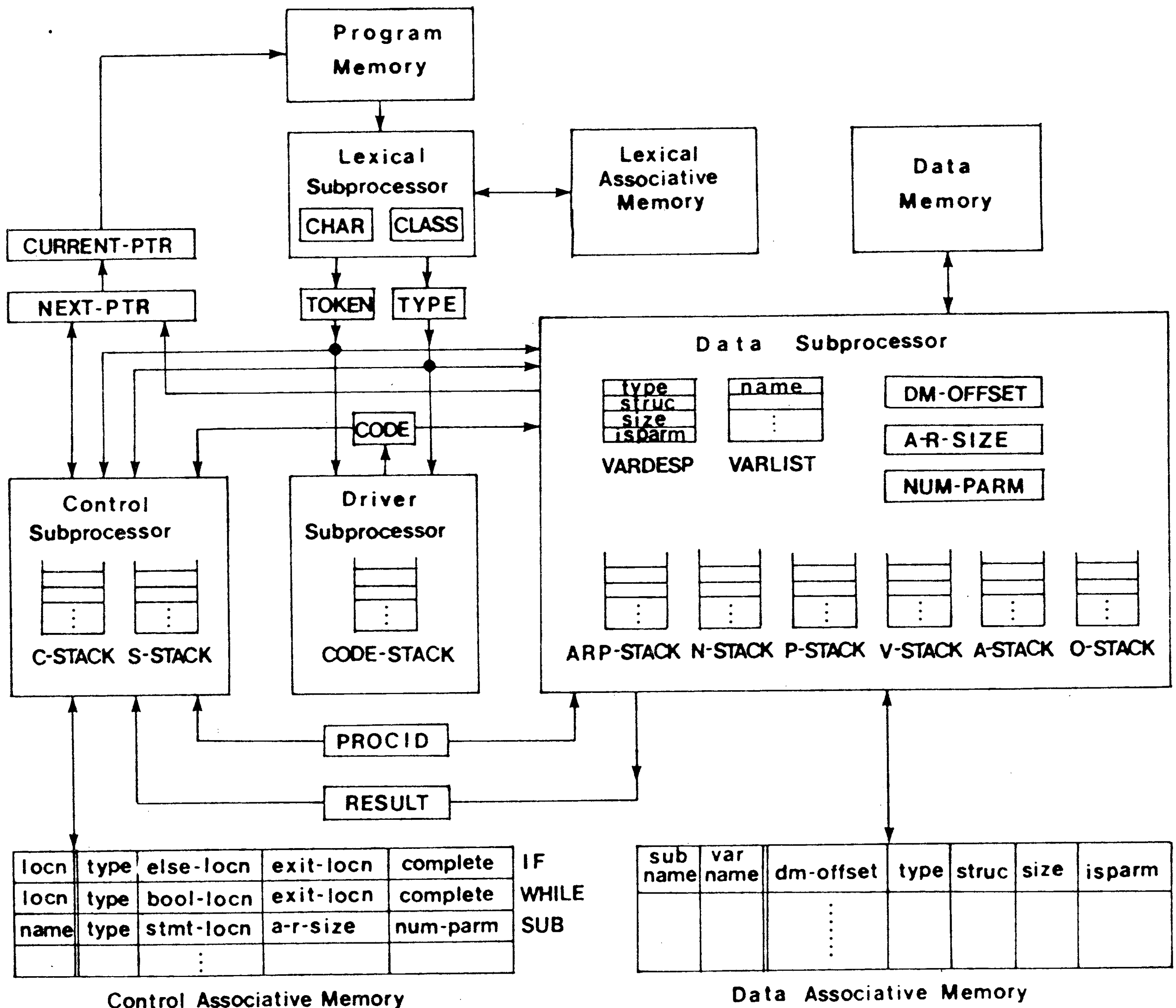


Figure 1. Detailed Hardware Organization of the Language Processor

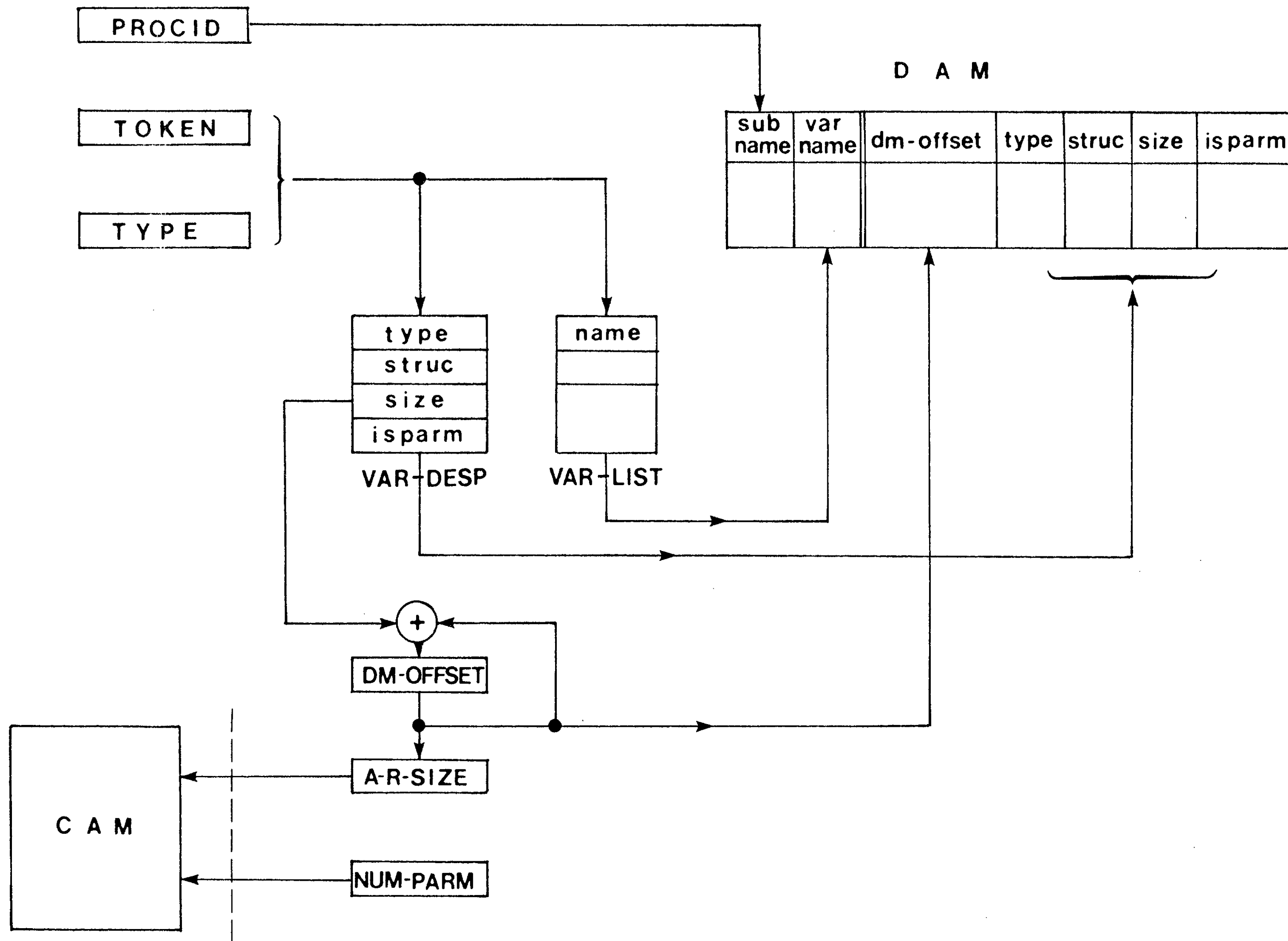


Figure 2. Data flow during the execution of declarations

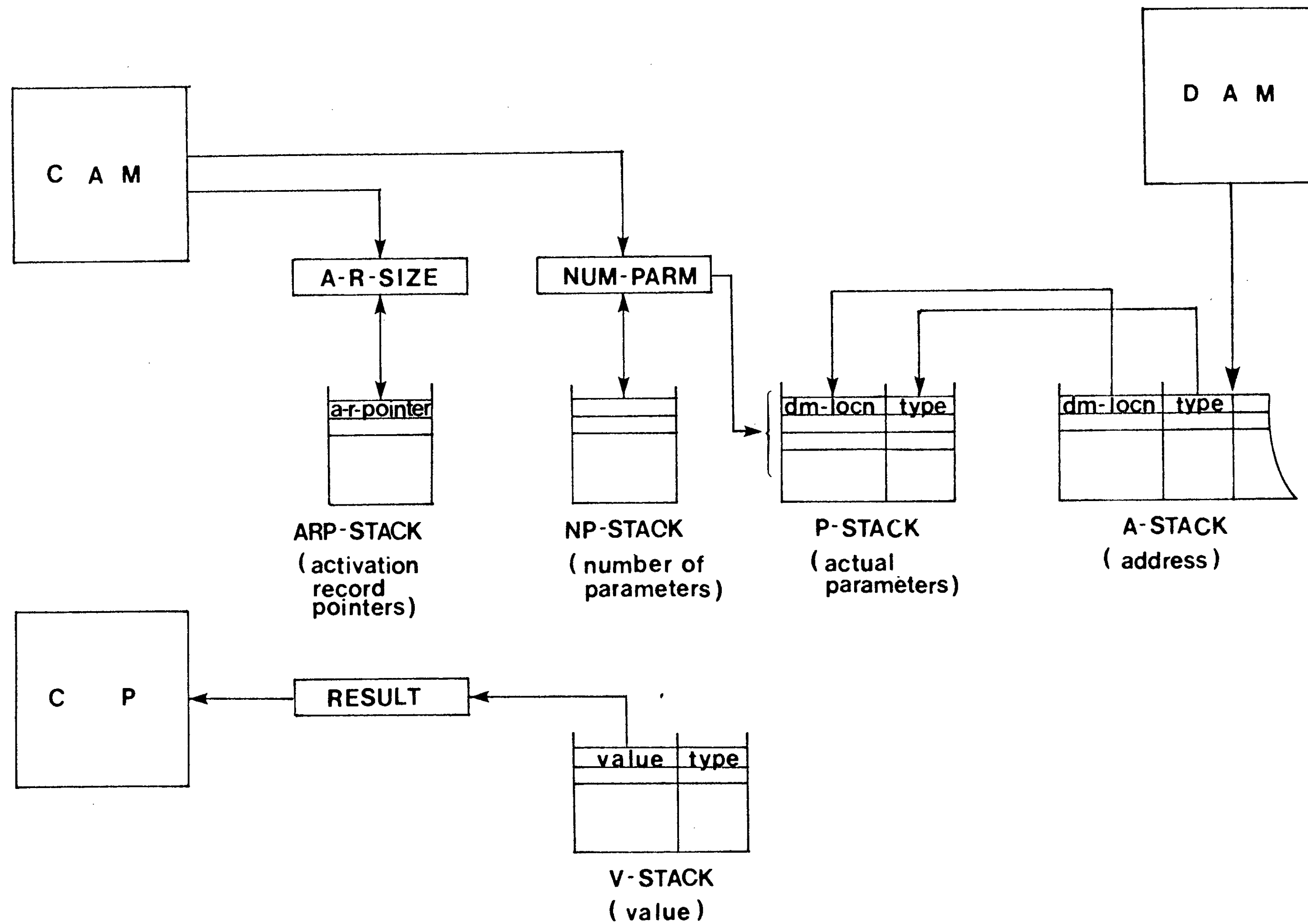


Figure 3. Data flow during the execution of subprogram call

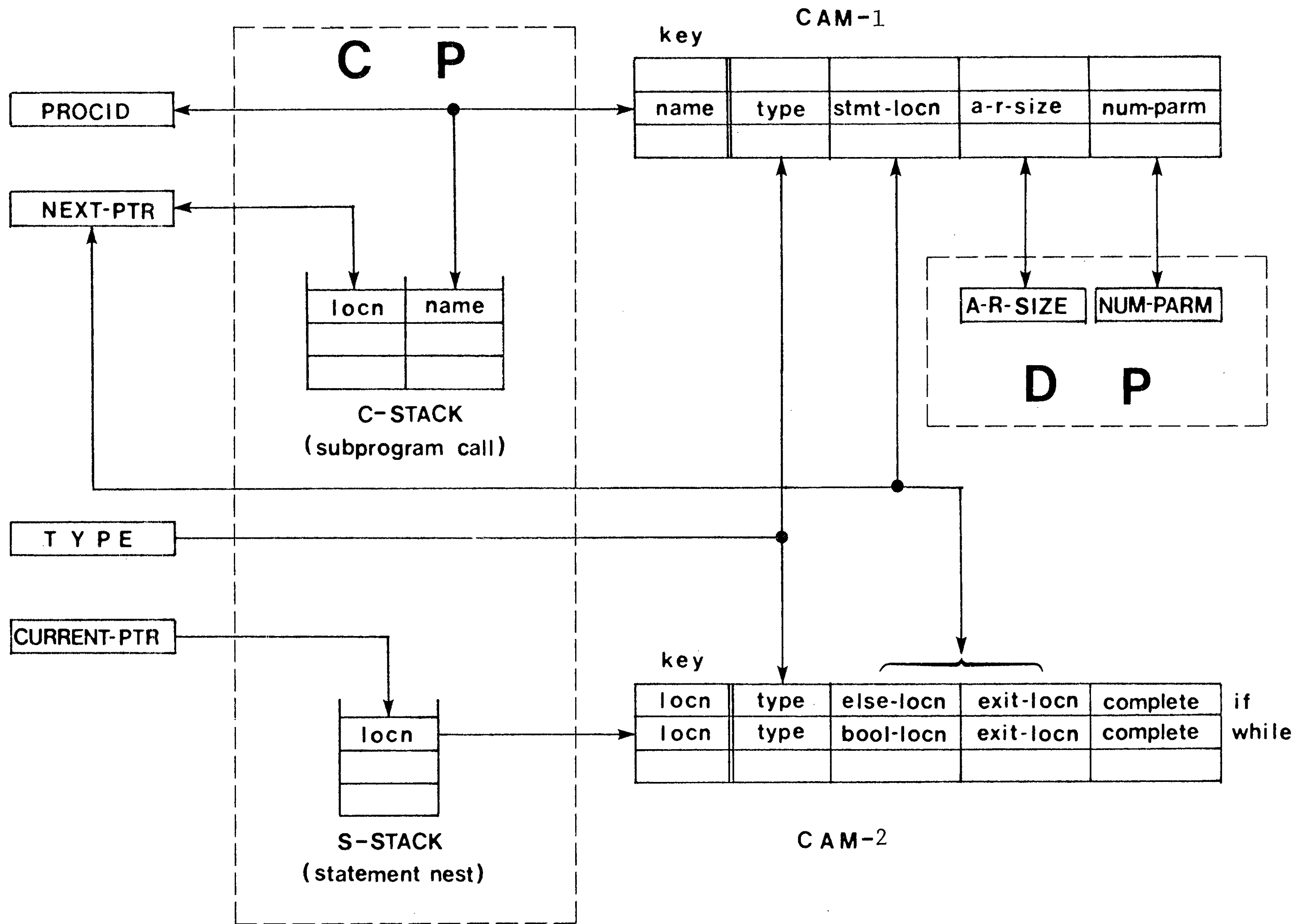


Figure 4. Data flow during the execution of control constructs

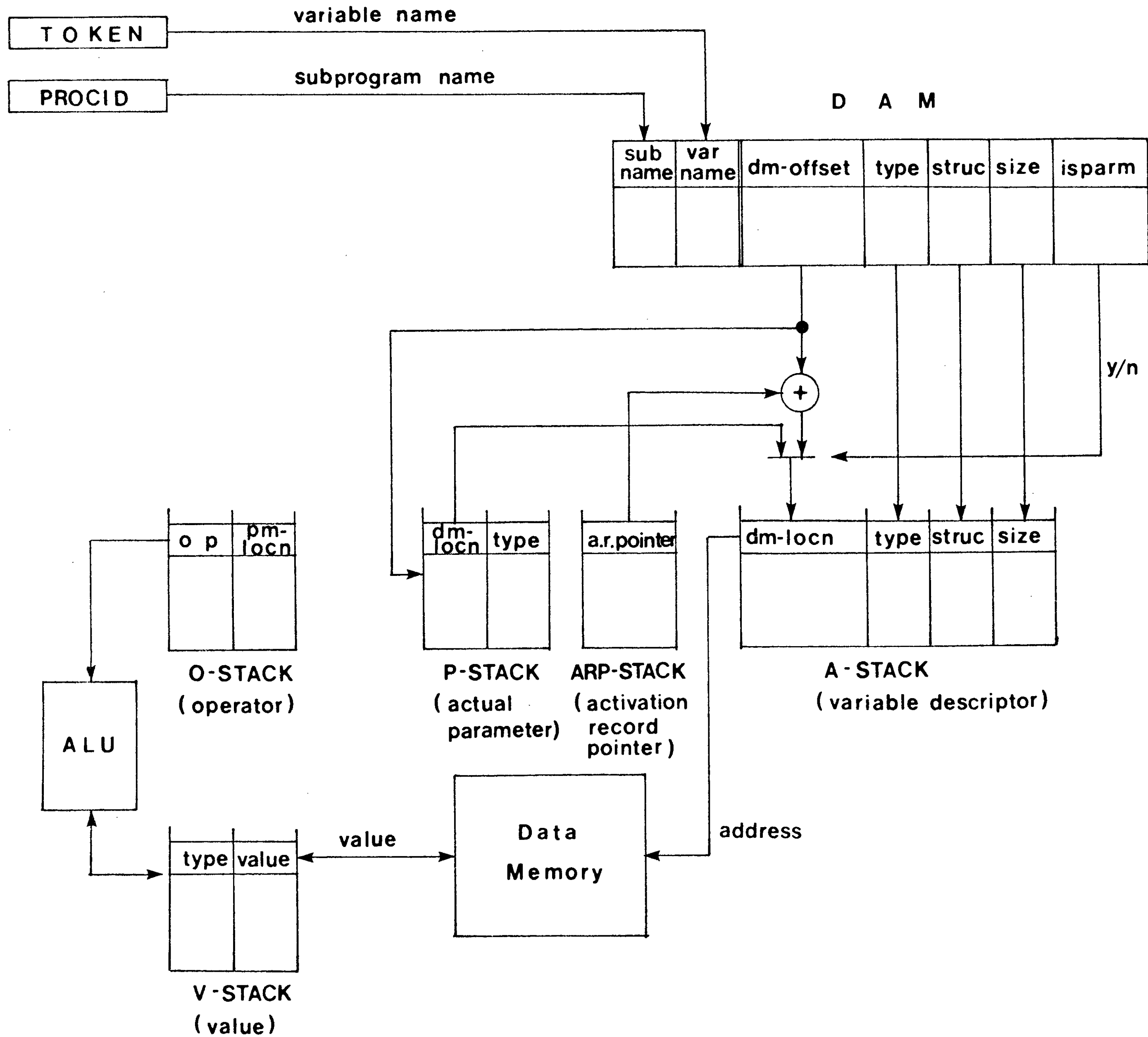


Figure 5. Data flow during the execution of assignment statement

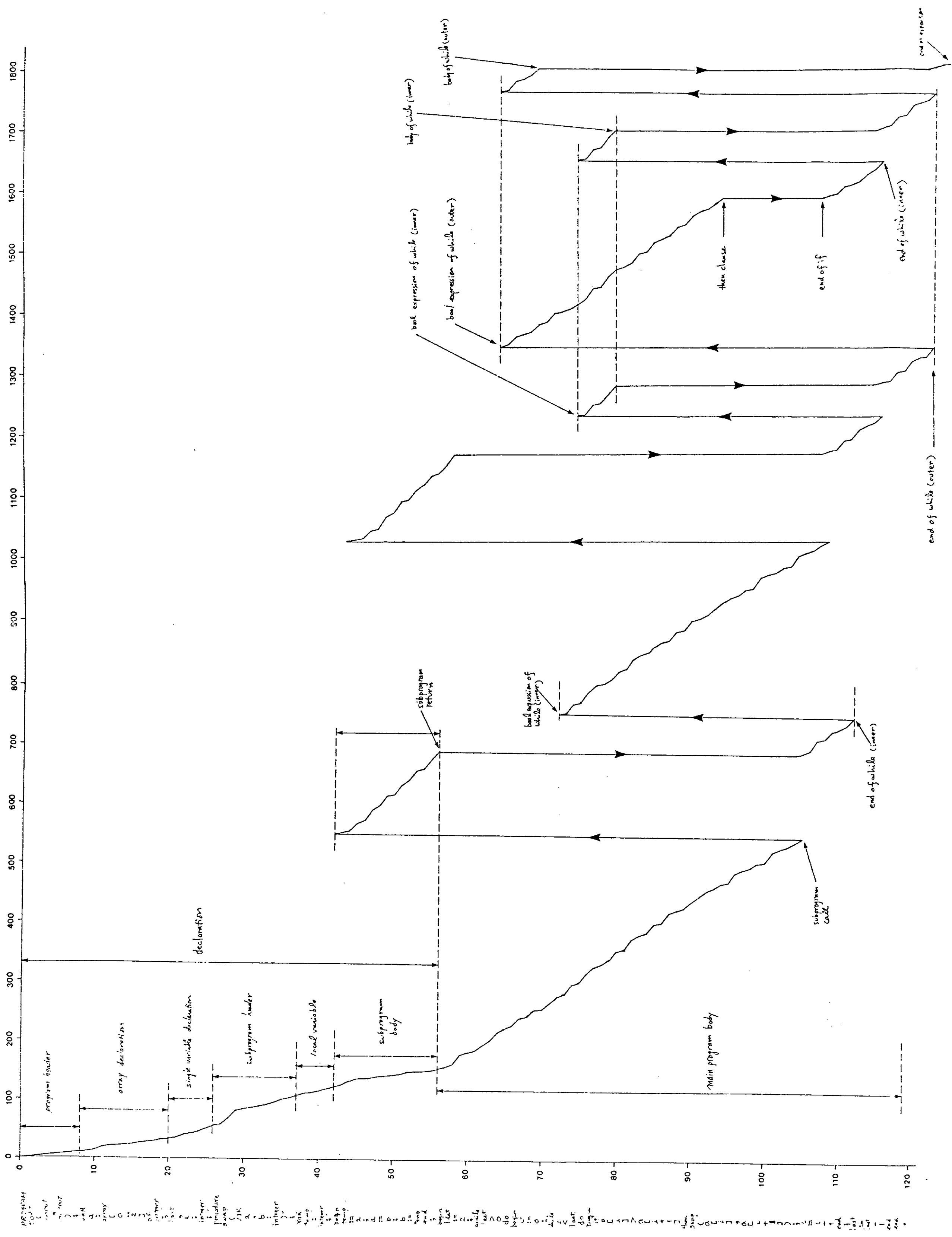


Figure 5. Simulation result for bubble sort program

INSTITUTE OF INFORMATION SCIENCES AND ELECTRONICS
UNIVERSITY OF TSUKUBA
SAKURA-MURA, NIIHARI-GUN, IBARAKI 305 JAPAN

| REPORT DOCUMENTATION PAGE | REPORT NUMBER<br>ISE-TR-83-34 |
|---|---|

TITLE

Performance of the Direct-Execution Computer PASDEC

AUTHOR(S)

Kozo Itano

| REPORT DATE | NUMBER OF PAGES |
|---|---|
| April 30 | 25 |
| MAIN CATEGORY | CR CATEGORIES |
| Register-transfer-level implementation | B5, B6, D3 |

KEY WORDS

High-level language computer architecture, direct-execution, register-transfer-level hardware design, PASCAL, hardware simulation

ABSTRACT

A Pascal interactive direct-execution computer (PASDEC) for a subset of Pascal has been designed and simulated at the register transfer level using the Computer Design Language CDL. In this paper, the direct-execution algorithm at the register transfer level is surmarized, and the execution performance of PASDEC is analyzed. The direct-execution time is given for a simple example, comparing it with the executions by the conventional microprocessors.

SUPPLEMENTARY NOTES