APL
PRESS
Box 378, Pleasantville
NY 10570 (914)769-3323

BULK RATE
U.S. Postage Paid
PERMIT NO. 28
Pleasantville
NY 10570

4

2 2

1977

APL NEWS

NOTICES

- As of June 1, the new business address of APL Press will be as shown on the cover page. Sales tax will no longer apply to orders from Pennsylvania, but will apply to orders from New York.

- Professor James England presented a paper on the use of APL in courses in Calculus and in Linear Algebra at the January meeting of the American Mathematical Society. Copies of the 12-page paper are available on request from the author at Swarthmore College, Swarthmore, PA 19081. The following is a brief extract from the introduction:

  > In the experimental courses we use computing in a central way: to represent the ideas and concepts of calculus and linear algebra and facilitate their analysis and exploration. To do this we use APL as the notation. In doing so we use the advantages of the APL notation to alter the presentation of a number of topics contained in calculus and linear algebra. The advantages we see in the APL notation are its relative conciseness, its lack of ambiguity, its ability to handle arrays in a straight forward manner and its demand for explicitly defined functions. A result of using the APL notation in these courses is a high degree of integration of the subjects of linear algebra and calculus.

- APL Press publications will be displayed at two forthcoming conferences:

  Ontario Universities Computing Conference
  University of Waterloo, Waterloo, Ontario, May 30-31

  Conference on Computers in the Undergraduate Curricula
  Michigan State University, East Lansing, MI, June 19-22

- Professor R. L. Spence is currently exploring the use of APL in teaching a Master's level course in computer-aided electronic circuit design, and invites correspondence from interested parties. His address is: Imperial College of Science and Technology, Department of Electrical Engineering, Exhibition Road, London SW7 2BT, England.

  The recent announcement of Dr. Spence's election as a Fellow of the IEEE cites him as "... a keen proponent of the APL notation" and as author of a textbook which uses APL as the exclusive mathematical notation (Resistive Circuit Theory).

- Jeff Shallit has been too busy with the work of his sophomore year at Princeton to produce a PLAY column for this issue. However, he submits a correction to the *CANON* function given in the last issue (`[3] Z←(∧\v≠A≠' ')/A`) and

Copyright 1977, APL Press

wishes to draw attention to the omission noted in the following letter from Robert A. Smith of Scientific Time Sharing Corporation:

> A bit of history was omitted from the APL-PLAY column in your Vol. 2, No. 1 issue. As some of your readers may recall, the Self-Reproducing Expression problem was published in the Problems Section of the Vol. 4, No. 2, January 1973 issue of APL Quote-Quad. The three solutions mentioned in your last issue can be found in the January and April 1973 Problems Section column.

- The many readers who have indicated an interest in APL implementations on micro-computers may expect to find several relevant articles in the July issue of Byte.

- Material for publication in the newsletter should be submitted in clean draft form for consideration. Book manuscripts should include a preface or introduction indicating the nature and scope of the material.

BOOK REVIEW

Garth H. Foster

Syracuse University

Gerrit A. Blaauw, Digital System Implementation, Prentice-Hall, Englewood Cliffs, N.J., 1976.

This book is a study of the implementation of digital computers, by one of the chief architects of IBM's 360 series. It gives the cleanest and most complete presentation of digital systems to be found in the literature. As such it begins with the architecture, proceeds to the implementation strategy and is completed in the realization. These three views of the structure at hand are taken in turn in the elaboration of the nine chapters and four appendices which comprise the book.

Throughout APL is used as an essential tool in the presentation of concepts and algorithms. APL is introduced piecemeal as needed and is nicely summarized in Appendix A. For those already familiar with APL this material may be easily skipped. For those less acquainted with APL, the breadth of the topics covered and the fact that the descriptions of the hardware may be executed should provide ample motivation for learning APL. The APL used is at the level of APL/360, but there appears to be no reason why the material will not run directly or with minor modifications on most APL systems.

The following observations were made while using this book as a text in a graduate level course, and they constitute more of an overview of the book than a critique of its contents.

Chapter 1 covers a number of preliminary items including the introduction of background information about System 360 which forms the basis of the running example throughout the text. Chapter 2 covers addition, and of course subtraction, of two's complement numbers. The architecture of addition is introduced and from it a bit adder is derived. The ripple adder leads to the carry-predict adder and its variants. Unfortunately the function *PREDG* of page 45 is not printed and the function *PRED* from page 43 is repeated. The correct version of *PREDG* as supplied to this reviewer by Dr. Blaauw is given below:

```
    ∇ C←G PREDG5 T
[1]   ⍝ PREDICTED CARRY FOR GROUPS OF 5
[2]    C←(⍴T)⍴0
[3]    C[;4]←CING5
[4]    C[;3]←G[;4]∨T[;4]∧CING5
[5]    C[;2]←G[;3]∨(T[;3]∧G[;4])∨∧/T[; 3 4],CING5
[6]    C[;1]←G[;2]∨(T[;2]∧G[;3])∨(∧/T[; 2 3],G[;4])∨∧/T[;
       2 3 4],CING5
[7]    C[;0]←G[;1]∨(T[;1]∧G[;2])∨(∧/T[; 1 2],G[;3])∨(∧/T[;
       1 2 3],G[;4])∨∧/T[; 1 2 3 4],CING5 ∇
```

In addition to the speed up obtained by replacement in the calculation of the carry path, the carry save adder is also of importance as it will occur in later chapters.

Chapter 3 deals with multiplication and after introducing multiplication of positive integers, problems of dealing with signed integers lead to recoding the multiplier and speeding up the process. A variety of techniques are introduced offering a wide range of speeds. *TWOMPY* which performs two bit shifts appears in later examples.

At this point Blaauw suggests that Chapter 4 can be skipped to take up Chapters 5 and 6, at least in part, and then one may return to Chapter 4 which deals with division. Once again a variety of methods are discussed at the implementation levels. for example, *ONEDIV* develops the quotient one bit at a time using non-restoring division on signed operands. Although there are more sophisticated techniques discussed in this chapter, *ONEDIV* is the division method selected for elaboration as part of Chapters 5 and 6 in the development of the data path.

Chapters 5 and 6 are closely related in that the data path is covered in Chapter 5 and control of the execution box is considered in Chapter 6. The first two sections of Chapter 5, which develop a data path for add/subtract and multiply, may be covered and then one may move onto the first section from Chapter 6 which develops an implementation for horizontal microprogram control of the data path. Attention can then be returned to the development of a common parallel data path and then to consider various alternatives in reducing the number and/or width of the data paths. When attention is

again returned to Chapter 6, the divider is used as an example of how control can be moved from the microprogram directly to hardware. Implementation strategies for reducing the width of the microprogram word leads to introduction of encodings in the fields of microprogram words and to vertical microprogramming. By the conclusion of these two chapters control of the execution box and its data path are considered down to the gate control level.

Chapter 7 turns to system control and the development of the I-box functions for instruction sequencing. Architecture, implementation, and gate control programs are developed, and while examples are restricted to register (RR) and register-indexed storage (RX) formats, sufficient detail is given to permit understanding the system implementation problems. Chapter 7 concludes with coverage of topics of encoders, decoders, and priority, including the least recently used (LRU) algorithm.

Chapter 8 discusses storage including topics of modular construction of storage, one level memory, and problems of address translation, hierarchical memory in the form of a cache, and look-aside memory for address translation. Topics of storage access, accessing data which is not aligned on a memory boundary, and pipelining conclude the Chapter.

Chapter 9 deals with communication between peripheral units and the rest of the system over a channel and device adapters which interface the peripherals to the channel. There is a wide range of peripheral device types and speeds and often the same peripheral device may be interconnected to different computing systems with widely differing characteristics. We should not be surprised that much of this chapter remains at the architectural level.

Appendix B gives a two page summary of System 360 instructions and a one page APL description of the Intel 8080. Appendices C and D give a summary of machine arithmetic and switching algebra respectively. The APL enthusiast will note a missing left (extra right) parenthesis on p.333 and a mislabeled table on p.352 (NAND is expressed incorrectly in APL).

The 142 problems distributed throughout the chapters are instructive and expand upon the range of examples. For instance, in the chapter on addition, adders for the PDP 11/45 and CDC 6600 are considered. The chapters on multiplication and division echo these cases as well as provide exercises on alternatives to the text strategies.

Chapter 5 has the TI SN74181 as an example (problem 5-3, p.181). Unfortunately there appears to be flaws in the function IMP74181 in that the data book from TI indicates that function selection is on pins 3,4,5,6 and not as indicated in line 3. Line 18 is in error as the output is on *POST*[13 11 10 9] and not *POST*[12 11 10 9]. Pin 12 is a ground as indicated in the comment on line 26.

In a similar manner the problems from Chapters 6 through 9 extend the topics covered by asking the reader to develop aids in understanding or verifying earlier material, or to consider alternative implementations. There are five extensive review exercises, including the investigation of floating point architecture, at the end of Chapter 9.

An errata sheet supplied to this reviewer by the author contains mostly typographical errors. These may mar the perfection of the book in detail but they should not detract from its practical use. Except for a missing close bracket at the end of line 1 of OPERATION on p.138 APL functions appear to be listed correctly.

As the physical size of computing systems continues to shrink, and as we become more involved with the pitfalls of programming, we need to be reminded how wonderfully complex a computer is. Professor Blaauw's book not only exposes those inner workings but also shows how complexity and cost may be traded for speed in the implementation.

ON DERIVATIVES

In Chapter 8 of his Calculus in a new key (APL Press, 1976), Orth presents a set of APL functions (represented in about a dozen lines of type) which precisely formalize the chain rule and the other rules for differentiation commonly taught in calculus courses. When entered in an APL computer these functions provide automatic differentiation of complex expressions. They also provide a clear and complete exposition of the differentiation process. The following excerpt illustrates the approach used:

8.2 A DERIVATIVE FUNCTION

We will now define a monadic function $\underline{D}$ such that:

The arguments of $\underline{D}$ are to be vectors of characters representing functions;

The values of $\underline{D}$ will also be character vectors representing functions. The function represented by a value of $\underline{D}$ is to be equivalent to the derivative of the function represented by the corresponding argument of $\underline{D}$.

The function $\underline{D}$ is defined recursively in terms of the derivative rules. For example, consider the monadic scalar function represented by the character vector '(1○⍺)+2○⍺'. The derivative of this function can be produced in terms of the dyadic plus rule. Evidently, the derivative is identical to the derivative of the function 1○⍺ plus the derivative of 2○⍺. Consequently it must be true that the function represented by the character vector

```
D '(1○⍺)+2○⍺'
```

is equivalent to the function represented by the character vector defined as follows:

```
'(',(D '1⍺'),')+',D '2⍺'
```

If we are to use the derivative rules in the definition of the function *D*, we must first define a function which determines the appropriate rule to be applied in each case. The selection of a derivative rule is based on the factorization of the function whose derivative is being produced. That is, in selecting a derivative rule to be applied to a function *H* we first determine a primary scalar function f and either one monadic scalar function *G* or two monadic scalar functions *F* and *G* such that either:

```
H ⍵    or    H ⍵
fG ⍵         (F ⍵)fG ⍵
```

In the first case we apply the monadic f derivative rule and in the second case the dyadic f derivative rule.

In either form of the factorization of *H* described by the above identities, *H* is equivalent to an expression such that whenever this expression is evaluated, the primary function f is evaluated last. Thus to determine an appropriate derivative rule for a character vector that is an argument of *D*, we can determine the symbol (and its location) of a primary scalar function that could be evaluated last whenever the function represented by this character string is evaluated.

In any expression without redundant outside parentheses, as in (*A*+*B*), the only candidates for the last evaluated function are those which are not imbedded within parentheses. For example, when evaluating the expression

```
((A+B)*C)-A×(B÷C)
```

the function ÷ is evaluated before × and + is evaluated before *, which is in turn evaluated before -. Finally, × is evaluated before -, so that - is the last function evaluated. The function

```
LP←'('
RP←')'
DEPTH:+\(⍵=LP)-0,¯1↓⍵=RP
```

can be used to determine those primary functions in an expression which are not imbedded in parentheses. *DEPTH* is a monadic function whose arguments are character vectors and whose values are vectors of integers indicating the depth of imbedding within parentheses of each character in the corresponding argument. For example:

```
   DEPTH '((A+B)*C)-A×(B÷C)'
1 2 2 2 2 2 1 1 1 0 0 0 1 1 1 1 1
```

The characters at depth 0 can be produced as follows:

```
      S←'((A+B)*C)-A×(B÷C)'

      (0=DEPTH S)/S
-A×
```

The primary scalar functions at depth 0 can be produced as follows:

```
      ((0=DEPTH S)∧S∈'+-×÷*')/S
-×
```

Among the primary functions in an expression, the one that will be evaluated last is the one at depth 0 which is furthest to the left. The index of this function in the character vector representing the expression can be produced as follows:

```
      ⎕←I←((0=DEPTH S)∧S∈'+-×÷*')⍳1
9
      S[I]
-
```

The primary scalar functions with which we will be concerned for the derivative algorithm are +, -, ×, ÷, *, ⍟, and ○. The index of the last evaluated function from this list in an expression is produced by the monadic function

```
      PFS←'+-×÷*⍟○'
      PF:((0=DEPTH ⍵)∧⍵∈PFS)⍳1
```

For example:

```
      PF S
9
```

and

```
      ⎕←I←PF '(10⍺)×20⍺'
5
      '(10⍺)×20⍺'[I]
×
```