Composite/Structured Design, Glenford J. Myers, Van Nostrand Reinhold Company, New York, New York, 1978. 174 pp. (ISBN 0-442-80584-5, $15.95).

It is well known that not only are design errors more frequent than coding errors in the production of software, but that they also take more time to be detected and corrected. It is the thesis of this book that a proper design methodology can produce programs of higher reliability and extensibility. The author presents an impressive set of guidelines and principles to aid in the design of medium- to large-sized programs, as well as examples that show proper application of the methodology. Most chapters in the book contain exercises that test the reader's understanding of the material presented, and a complete set of answers is provided at the end.

The key idea of composite design is to reduce complexity by maximizing module independence. This is achieved by maximizing *module strength* (a measure of the internal relationships of a single module) and minimizing *module coupling* (a measure of the intermodule relationships among all modules of a program). Two chapters are devoted to a discussion of different types of module strength and coupling. For example, the most desirable type of module strength is called *informational strength*; its purpose is to hide some concept or data structure within a single module. The most desirable type of module coupling is called *data coupling*, whereby modules communicate with one another by means of input and output arguments only.

Composite design also comprises a methodology of problem decomposition. Three types of decomposition are identified and described in some detail in a chapter devoted to each. A slightly different and interesting approach—called the Jackson design method—is also discussed, but its relationship with the previous decomposition techniques is not well analyzed at this time.

The ideals of module independence and selection of decomposition techniques are illustrated in an excellent analysis of a specific application problem. Finally, six popular programming languages are compared for those features that support the proposed design methodology.

In the opinion of the reviewer, the design methodology presented in this book should become part of the intellectual equipment of every programmer and system designer. It is reasonable to assume that this approach will be further refined and analyzed in the years ahead. In particular, many of these ideas should be supported and encouraged by features and concepts that are finding their way into new programming languages.

B. Leavenworth

Advances in Computer Architecture, Glenford J. Myers, John Wiley & Sons, Inc., New York, New York, 1978. 314 pp. (ISBN 0-471-03475-4, $21.00).

Has the logical organization of the computer CPU changed very much in the last fifteen years? Not much, the author of this book suggests. If a first course in computer organization concentrates on the design ideas of fifteen years ago it will also be describing the workings of most contemporary machines. If a textbook for a second course, such as this one, presents more advanced ideas, it will be presenting concepts that have not yet gained acceptance in the current computer market.

About half of the book is devoted to a case-history approach that describes in some detail the organization of four machines that contain the ideas of interest. The first part of the book introduces the main ideas, such as machine implementation of expression-evaluation stacks, self-defining data and higher-level addressing, and argues their importance. These techniques are needed to close the "semantic gap" between many of the notions of high-level languages and the concepts of most languages. For example, PL/I has a variable-length string, and System/370 does not. Also, System/370 has general registers, but PL/I gives no way to manipulate them. At present, compilers and interpreters must bridge the gap, at a significant cost in computer performance and in program development effort.

Two of the machines described exist on paper only. The first, Student PL Machine (SPLM), serves very well as a tutorial device to tie the concepts together. The other "paper" machine—the subject of the author's Ph.D. thesis—is a design whose primary objective is to improve the reliability of programs. This is accomplished by providing more hardware checks, fewer opportunities for program ambiguity, and better testing facilities.

Another machine, termed the SYMBOL processor, is a one-of-a-kind machine and an excellent study of the hardware implementation of a rather high-level-language computer. The only description of a machine in widespread use is that of the Burroughs B1700. This system is capable of providing several different user interfaces by changing the microcode of this versatile underlying machine. Unfortunately, the details provided on the B1700 are skimpy.

All the descriptions are very well done. First, the highlights are presented; then an example activity is sketched to show how the pieces function together; and, finally, the details are supplied. (Why aren't manufacturers' manuals written like this?) The book closes with a discussion of the main objectives of a computer ar-

chitectural design. The author shows how studies of various systems and encoding methods might be used to refine a system architecture.

This book will make a good advanced textbook. Although the concepts are not brand new (three of the described machines predate 1974), they are clearly defined and motivated. Perhaps the best influence this book might have would be to encourage readers to elaborate and extend the key ideas of the book: to discover how data and algorithms can be more effectively represented; to find better mechanisms underlying program structures; and to gain a clearer understanding of the tradeoff between hardware and software.

L. Haibt

The editors assign reviews of books that might interest our readers. Reviews are signed, and opinions expressed are those of the reviewers.