Software standards—With hints of their relation to computer architecture

by H. HECHT
Consultant
Los Angeles, California

INTRODUCTION

Software standards can help the developer, the user, and regulatory agencies in facilitating communications, in extending software life, in broadening the applicability of programs, and in reducing the personnel and organizational reorientation required in transitioning from one project to another.

Communications are facilitated by use of standardized terminology, by use of standards for development methodology (e.g., structured programming), and by standardized test reporting to name just a few pertinent areas. Facilitating communications implies not only that misunderstandings are avoided but also that the software procurement cycle can be shortened and procurement costs reduced because both contracting parties may from prior experience know their obligations when a standard is invoked. Use of a standard also reduces the cost of software development because the allowance for coping with unfamiliar specifications can be reduced.

Software standards can extend the useful life of programs by facilitating adaptation to new versions of operating systems, by making it possible to transfer the program to a new computer, and by making maintenance and updating of the program itself a lot easier and cheaper. The broadening of a program’s applicability is made possible by the features cited above plus uniformity among users of program format, test, and test reporting requirements that can be accomplished by standards.

Much time is required during the start-up phase of critical programs in order to train otherwise experienced analysts and programmers in specific contractually imposed design, coding, or documentation requirements. Allowances have to be made for errors in complying with unfamiliar requirements, and at times management may be reluctant to bid on a software development that is within its normal range of business interests simply because of uncertainty regarding a test or documentation requirement that had not previously been encountered. Software standards could reduce this wasted effort in transitioning from one project to another.

In view of these benefits one must ask why the few existing standards are not more widely employed and why there is not more effort at implementing software standards.

Part of the answer lies in the newness of software engineering as distinct from programming for one’s own use. The latter implies the translation of a defined sequence of operations into machine-readable form, an activity that was best managed in a local context and to which standards are about as applicable as they are to the writing of business letters. Software engineering recognizes that requirements analysis, test, validation, program maintenance and related tasks, in addition to programming proper, are necessary to provide the computing service that the user expects and needs. This broader activity involves the interaction of many groups, extends over a long period of time, and is in scope quite like a construction project. Hence, standards are appropriate to guide the activities.

But together with the recognition of the benefit of standards must also come the recognition that they involve costs. A standard for documentation may require a more detailed format than is essential for a given application, and a software quality assurance standard may impose access control that will be perceived as burdensome by personnel that arranged the mounting of their own test tapes as they saw fit. In the context of computer architecture it is appropriate to point out that just about every software standardization activity will increase memory and throughput requirements. For example, standards for test or test documentation may invoke analysis tools that slow down the execution of the primary program. So right at the outset let it be said that whatever can be done to increase the speed of computers and the size of the easily accessible memory will be welcome for the application of software standards.

In terms of scope, software standards can be divided into three broad categories: language standards, software engineering standards, and software implementation standards. Each of these classes will be separately discussed below. There is another important classification of standards by issuing agency: voluntary industry standards, semi-official standards (e.g., those promulgated by international agencies), federal standards, and military standards. This paper will be principally concerned with non-governmental standards.

Finally, the common use of the term standards encompasses some documents that are actually recommended practices, guidelines, or specifications. These documents
obviously have a lesser enforceability or more restricted applicability than true standards, but, in view of the rather limited effect of any software standards on computer architecture, these distinctions are not too important in the present context.

LANGUAGE STANDARDS

Standards promulgated by professional organizations are presently applicable only to High-Order Languages (HOLs). As such, their effect on computer architecture is indirect; clever compiler designers, if necessary, can cope with almost any machine instruction set. Military agencies may be motivated to standardize the assembly level instructions for certain classes of computers, primarily those that process tactical software. Implementation of such standards would have a much more direct effect on architecture, but discussion of this event is beyond the scope of this paper which emphasizes voluntary compliance.

We should also be careful to point out that the effect HOLs on computer architecture is to a large measure independent of the existence of language standards. Thus, the adoption of features such as general register organization and floating-point processors that are a boon to processing FORTRAN object code preceded the adoption of the ANSI FORTRAN standards. Standardization of HOLs may nevertheless be significant in this connection because it increases the portability of code and makes the benefit of architectural features that support HOLs more visible to the user. Since source code written in a standardized HOL can be executed on any computer for which a compiler written to that standard exists, it can become quite apparent to a user interested in such matters how both compile time and run time are affected by the computer organization. Hence, standardization of HOLs in general is a positive factor in promoting architectural features that support processing of HOL-derived code.

So far the discussion has dwelled on use of compilers for translating a program coded in a standard HOL for generating the instructions that ultimately are operated on by the computer. Another approach for handling HOL source code is through direct-execution machines. Considerable research in this field has been under way for some time so far leading to full-scale development of only a few direct-execution computers. Among many factors responsible for the slowness of this development may have been the proliferation of language "dialects", requiring translation or other special processing before existing source code could be processed on a direct-execution machine. HOL standardization may remove at least this factor among the obstacles to acceptance of architectures for direct execution.

Block-oriented languages, such as ALGOL, benefit particularly from provision of stack features in the computer. Further standardization of block-oriented languages may therefore become a factor in promoting stack architectures.

The general tendency toward increased use of standardized HOLs (likely to be heightened by a recent DoD directive) will tend to benefit those computers that can efficiently process code derived from them. However, it must also be realized that going from assembly code to HOLs will require more memory and greater processing speed if the same computational capabilities are to be provided. Thus, the previously mentioned effect that standardization demands faster computers with larger memories is evident in the area of language standards.

SOFTWARE ENGINEERING STANDARDS

In the broader sense, dictionaries for information processing that have been available for some time could be considered software engineering standards. Also, some Department of Defense acquisition management documents contain material related to software engineering standards. But specific software engineering standards are only in the draft stage, e.g., a Military Standard for Tactical Software Development was circulated as a draft in June 1977. It is also interesting to note that the Federal Information Processing Index provides for software engineering subjects, e.g., operating procedures and operating systems, but that there are at present no entries under these headings.

Effectiveness of software procurement and development suffers, and in some cases the introduction of computers for critical tasks is being hampered, by this lack of standards. The IEEE Computer Society Subcommittee on Software Engineering Standards is now active in preparing three standards in this area: software development, software quality assurance, and software test documentation. A specialized terminology for software engineering has also been drafted by this committee.

The software development standard is expected to require a modular organization, some form of structure programming, and "readable code." These requirements do not easily translate into effects on machine architecture but they will in general result in an expansion of code over that generated without them and hence again will demand greater memory capacity and speed.

Standards to be drafted for the quality assurance area may, among other things, be concerned with controlling access to software and with recording of all executions of a program during test or certain other critical phases. In current practice compliance with such requirements can be monitored by making use of features of advanced operating systems. As a result of the standardization the requirements for such controls may become sufficiently common that it may be profitable to explore architectural features that can simplify the demands placed on the operating system. In the quality assurance area there is an existing Military Specification on Software Quality Assurance Programs that has no apparent impact on computer architecture.

Test documentation standards will be concerned with areas such as reporting completeness of test, complete description of the test environment, and accurate recording of all changes to software or the environment over the usually rather extended period required for test of critical software programs. One of the methods for reporting completeness of test, but probably not the only one that would be acceptable under
such a standard, is the recording of all node exits taken during a series of test runs. In current practice this can be done by use of static analyzers to locate the nodes, and subsequent "instrumentation" of the nodes such that passage during execution can be counted. Execution time of such an instrumented program can be orders of magnitude longer than that of the original code, and the need to insert and then to remove the instrumentation may impair validation of the test. This is one of the major obstacles to more widespread use of these software test tools, and architectural features could possibly be brought to bear to simplify instrumentation requirements and to reduce the execution time penalties.

STANDARDS FOR SOFTWARE DELIVERABLES

Software can be delivered to the user in a large number of ways: installed in a specified computer, as a listing, as card decks, as paper tape, and as magnetic tape to a number of formats. Within most of these there are further variations on character set, density, and other format variables. Physical dimensions of the deliverables and format features are, in many cases, covered by standards or trade practices. In most cases these affect the design of computer peripherals much more than computer architecture proper. As a matter of fact, there seems to be a tendency, through the use of microprocessors, to make peripherals "smarter" (i.e., to adapt themselves to format variations) and thus to remove the need for some of the conversion tasks that must now be handled by processing in the central computer.

Software documentation is an important deliverable that is covered by a number of voluntary standards. In addition, this area is the subject of numerous Department of Defense specification and data item descriptions (DIDs). None of these documentation standards appears to have an immediate or even remote impact on computer architecture.

CONCLUSIONS

The imposition of software standards will, in several areas, require longer code (i.e., more storage) and slower execution (i.e., require a faster computer in order to handle a given task). Thus, software standardization imposes a requirement for more powerful computers. It would, however, be probably more correct to say that the capability of industry to provide more powerful computers permits standardization rather than the other way around.

The requirements for HOL processing, in particular the requirements of newer HOLs, may favor certain architectural features in the central processing unit, and the fact that these languages are standardized may impose a further urgency on the computer community to adopt architectural features that support these languages.

Software engineering standards may become accepted in the near future. These will require some code expansion, control of software access during critical test phases, and may lead to more widespread usage of automated test tools to quantify completeness of test. Requirements likely to be formalized by these standards are currently being met by software but hardware features to support the software or to supplant it may at some future time be worth investigating.

Standards for software deliverables primarily affect peripherals. Some unburdening of the central processing unit by "smart" peripherals may be in the offering. Standardization of output formats may further simplify the central processing requirements in servicing peripherals.

REFERENCES

1. Basic FORTRAN ANSI X3.10—1966
2. ANSI FORTRAN ANSI X3.9—1978 (also, ANSI X3.9—1966)
4. Programming Language COBOL ANSI X3.23—1974
5. Programming Language PL/I ANSI X3.53—1976
6. Minimal BASIC ANSI X3.60—1976
7. Programming Language Minimal COBOL ANSI X3.65—1977
10. Computer issue on Stack Machines, May 1977