CGTM NO. 134



# THE SOFTWARE CONTEXT OF MICROPROGRAMMING

John V. Levy

Computer Science Department
Stanford University
and
Stanford Linear Accelerator Center

Presented at a one-day short course in Microprogramming, University of Santa Clara, Calif., June 24, 1972

## 0. Introduction

Microprogramming is programming; the primitive operations with which the microprogrammer works are just concerned with more detailed aspects of the computing system. The first section of these notes will attempt to paint a broad-brush picture of software and programming in general. This context will enable us to see where microprogramming fits in among the activities which create a general-purpose computing system. Another reason for looking at the whole picture is that the trend, especially in special-purpose systems, is to do away with some of the levels of software which used to be "necessary". In order for the system designer to integrate some of these levels, he must understand what functions they performed.

The second section deals with the relation between computer design and microprogramming. The "architect" of a compatible series of computers now must worry about many things other than gate-level design.

In the last section, we will try to see how some hardware design principles - and economics - are creeping into software design; and how some software tools are being used in hardware design.

## 1. Programming Languages

Figure 1 shows four "levels" of activity which are going on when a program is running in a microprogrammed computer processor. At the far left, the flowchart of the system analyst indicates only the very general concepts which are necessary to the solution of a problem. Here, the task is to read in a series of numbers, sort them, and then write them out. The next column shows how the programmer has implemented the box marked SORT, using the PL/I programming language. The programmer has made decisions about how the data is arranged in memory and about which algorithm to use to perform the sorting. (The method shown here is the "bubble sort" method which is, in fact, very inefficient for large tables.)

The next column shows a portion of the machine language listing for this same program, written for an IBM System/360. Another type of programmer could have created this level of program directly from the flowchart; but, in fact, these instructions were generated by the IBM PL/I compiler (F-level, version 5.1). At this level, the programmer thinks he is manipulating the computer hardware. As microprogammers, we know that this may or may not be true. Finally, the right-most column shows a small section of the micro-sequencing for an IBM System/360 Model 40 [from Husson, Reference 1]. This, at last, indicates what is going on at the

level of the registers and data paths of the hardware. The notation here is again a kind of flowchart which is from IBM's CAS (Control Automation System) language. The whole page from which this was taken is shown in Figure 2.

All of these levels are active at the same time. Each one represents the activity of the computer at a different degree of detail. Each also represents a level of programming, where a programmer has selected from a set of available operations to produce a sequence which, when executed, carries out a desired action.

Let's look at the process of compiling - the kind of process which generates machine language programs from "higher-level" language programs, such as ALGOL, COBOL, FORTRAN, or PL/I. Figure 3 shows the two steps required for running a program which is written in a language which must be compiled. First, the source program is read (as if it were data) by the compiler program, and the machine language object program is output. Then, the linkage editor and loader read the object program back in, load and make connections to the programs that make up the "run-time environment", and then "control is passed" to the object program - that is, it is allowed to execute. Compilers and loaders are themselves sophisticated packages of programs. And the run-time environment - which includes such things as input/output handling, program error conditions, data type converstion, and programmer debugging aids - is a complex set of routines which currently have to be developed by the implementers of the high-level language. Microprogramming, in the view of some programming language designers, holds the promise of developing computers in which the machine language is identical with the operations one wants performed in the run-time environment of a high-level language. For example, when a data converstion error interruption occurs, why not have a register (or a stack) in which the current statement number of the source program is easily available? And the operation of moving that number into an error message might be a machine language instruction (Figure 4).

The criterion of a good programming language design - at all levels - is, "does the language have natural constructs for the things programmers want to say?" If it doesn't, then no amount of good hardware design will be able to make up for the inefficiencies of the programs or microprograms.

## 2. Interpreting and Emulating

Looking back at Figure 1, we see that each operation at a given level is actually created by a series of operations at the next level to the right. When an operation to be performed is encoded as an operation code, stored in a memory, then later retrieved by a lower-level program which carries out the intent of the operation, this process is called interpretation. The great suggestion of Von Neumann, which led to modern computers, was that operation codes and data could be stored in the same memory. Now with microprogramming, we are coming to treat operation codes (programs) very much like data even at the machine language level. In Figure 1, there is only one example of interpreting - the microprogram is interpreting the machine language instructions. The higher-level language program, rather than being encoded directly and interpreted, was instead translated into machine language by the compiler.

Figure 5 shows the diagram of instruction fetching, decoding, and executing; when this process is implemented in microprograms, the decoding becomes, for example, a multi-way branch, based on the value of the operation code. The power of microprogramming is that there can be a tremendous difference between the functional characteristics of a computer instruction and the actual hardware used to carry out the instruction.

We use the word "computer architecture" to describe the functional characteristics of the machine as seen by the (machine language) programmer. "Computer organization" then refers to the actual physical hardware. For example, in the IBM System/360 architecture, there are 16 general-purpose registers. In the Model 20 processor organization, however, these registers are kept in core memory (Figure 6). In particular, then, microprogramming has allowed IBM and others to produce compatible series of computers which have a single architecture, but where the cost-performance tradeoffs have been made by varying the organization of the different models.

The availability of microprogramming has made another feature more easily available - backward compatibility by emulation. Emulation is the term applied to interpretation of a machine language for which the organization was not primarily designed. "Emulation" is no longer a distinctly different process now that a microprogrammed machine has been designed with no one specific architecture in mind (Lawson, Reference 2). The prospect of writable microprogram storage makes designers dream of having a different machine language for each application.

For example, while compiling, the compiler machine language would be active. Then, during object program execution, perhaps a computing-oriented instruction set, including floating-point operations, would be loaded in (Figure 7).

Hardware organization may restrict the efficiency of some architectural features of a machine, but it does not determine them. Microprogramming has allowed computer architecture to become more independent of the hardware organization. The specification of machine language details can now take place after the hardware structure has been determined. And thus, the computer architect is free to pay more attention to the demands of higher-level software functions.

## 3. Future Trends

The economics of large-scale integrated circuits has placed a premium on regularity of structure - many copies of a single chip are much cheaper than several different chips. Memory is the most regular-structured part of a computing system and microprograms make good use of fast memory. Thus, the technology is encouraging the developments indicated in the previous section.

The modularity of programs is also gaining currency as more single-application small computers are sold. The idea is to create software packages which can be pulled off the shelf and plugged together to implement a "custom" requirement. There is no reason why microprogrammed packages should not be part of such a development.

On the other side of the coin, some large software "packages" are being used now to automate the computer implementation process (Figure 8). Some of these are

- circuit design aids
- logic simulators
- chip and board layout programs
- wire list maintenance programs
- wire-wrap drivers
- whole-system functional simulators

Microprogrammed computers, since they are in general composed of simpler functional units, are particularly well suited to simulation approaches to design.

In addition, microprogram assemblers are gaining wide use, and considerable effort is going toward higher-level languages for writing microprograms.

## 4. Summary

Microprogramming, from the software point of view, is just another level of interpreting which allows machine languages to be more flexible. There is an incentive due to the cost of all programming, however, to make machine languages do more. This requires the computer architect - the designer of the machine language instruction set - to understand more about higher level language constructs and run-time environments.

Microprogrammed computers are advantageous for achieving

- compatibility with earlier models by emulation
- series of single-architecture models with varying performance
- later definition of machine language in the design process, and easier evolution and correction of instructions

A large body of software is in use as design aids for computers in general. Simulation is a particularly well suited tool for microprogrammed computer design.

## References and Bibliography

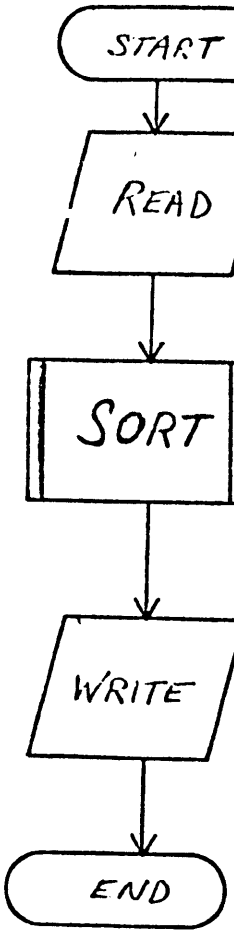

1. Husson, S.S., "Microprogramming Principles and Practices", Prentice-Hall, Inc., Englewood Cliffs, N.J., 1970.

2. Lawson, H.W., Jr., "Programming language oriented instruction streams", IEEE Trans. Computers, Vol. C-17, No. 5 (May 1968).

3. McKeeman, W.M., "Language directed computer design", Proc. AFIPS Fall Joint Computer Conference (1967), pp 413-417.

4. Rosin, R.F., "Contemporary concepts of microprogramming and emulation", Computing Surveys, Vol. 1, No. 4 (Dec. 1969), pp 197-212.

5. Stevens, W.Y., "The Structure of System/360, Part II - System Implementations", IBM Systems Journal, Vol. 3, Number 2, (1964), pp 136-143.




```
SORT1: PROCEDURE;
   DECLARE (TABLE1(1:32),I,J,N,TEMP)
   N=32;
   DO I=1 TO 32;
      TABLE1(I)=33-I;
      END;
   DO I=1 TO N-1;
      DO J=I+1 TO N;
         IF TABLE1(J)<TABLE1(I)
            THEN DO;
               TEMP=TABLE1(I);
               TABLE1(I)=TABLE1(J);
               TABLE1(J)=TEMP;
               END;
         END; /* J-LOOP */
      END; /* I-LOOP */
   END; /* SORT1 */
```

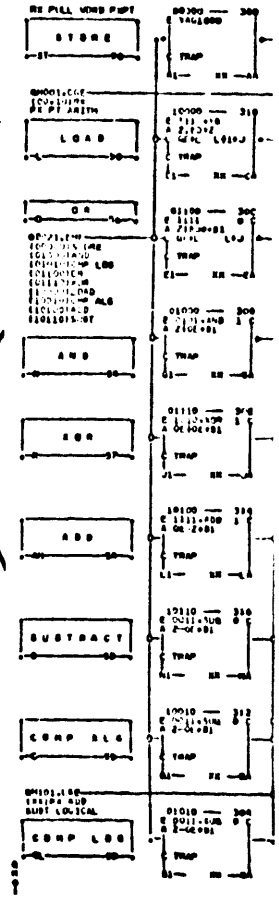
```
         LH     7,I
         AR     7,7
         LH     14,VC..TABLE1(8)
         CH     14,VC..TABLE1(7)
         BC     10,CL.18

         LH     8,I
         AR     8,8
         LH     14,VC..TABLE1(8)
         STH    14,TEMP

         LH     8,I
         AR     8,8
         LH     7,J
         AR     7,7
         LH     14,VC..TABLE1(7)
         STH    14,VC..TABLE1(8)

         LH     8,J
         AR     8,8
         LH     14,TEMP
         STH    14,VC..TABLE1(8)

CL.18    EQU    *
```



| PROBLEM DESCRIPTION | HIGH-LEVEL LANGUAGE | MACHINE LANGUAGE | MICRO-CODE |
|---|---|---|---|
| FLOWCHART | PL/I | IBM /360 ASSEMBLER | 360 MODEL 40 |

FIGURE 1

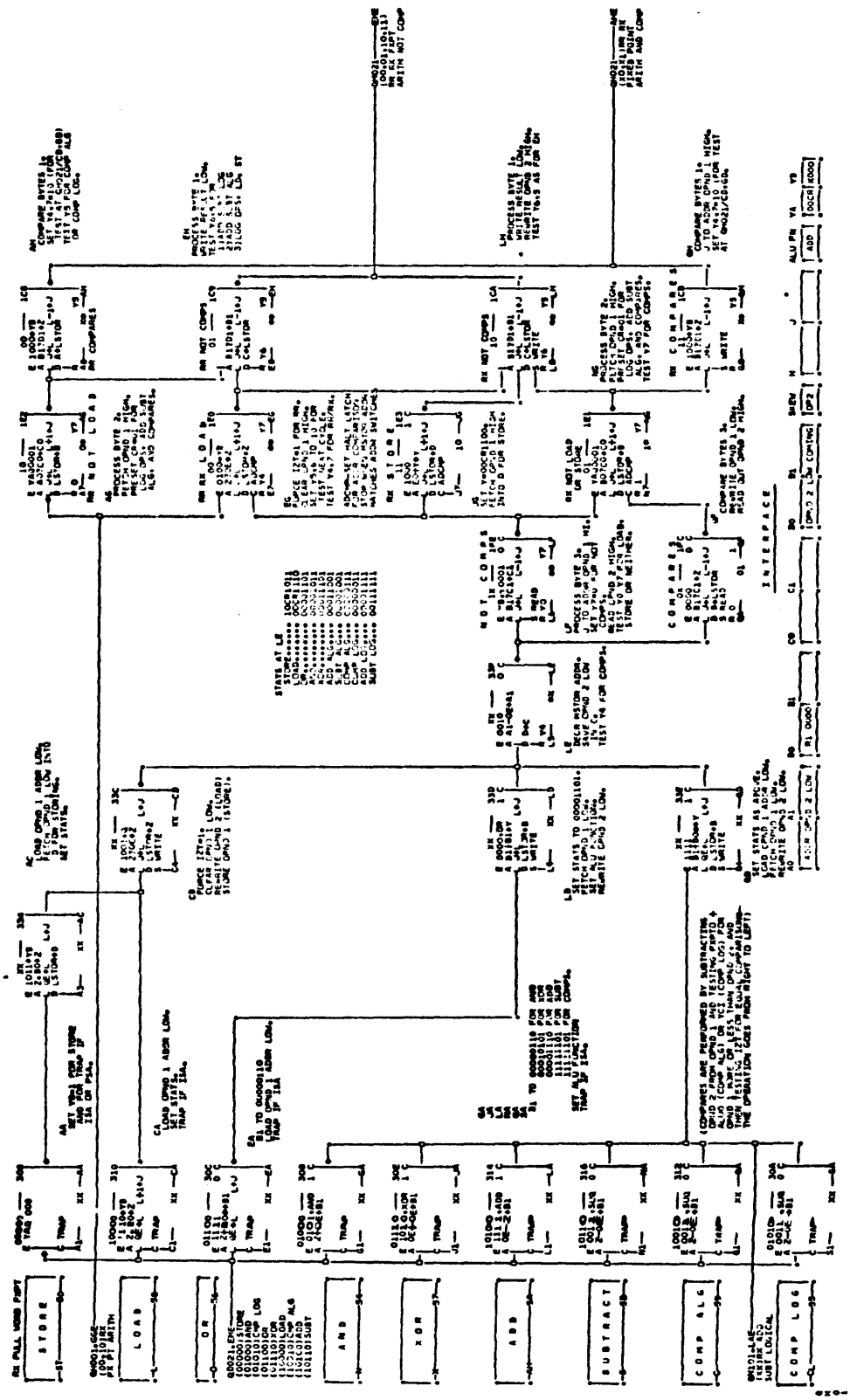
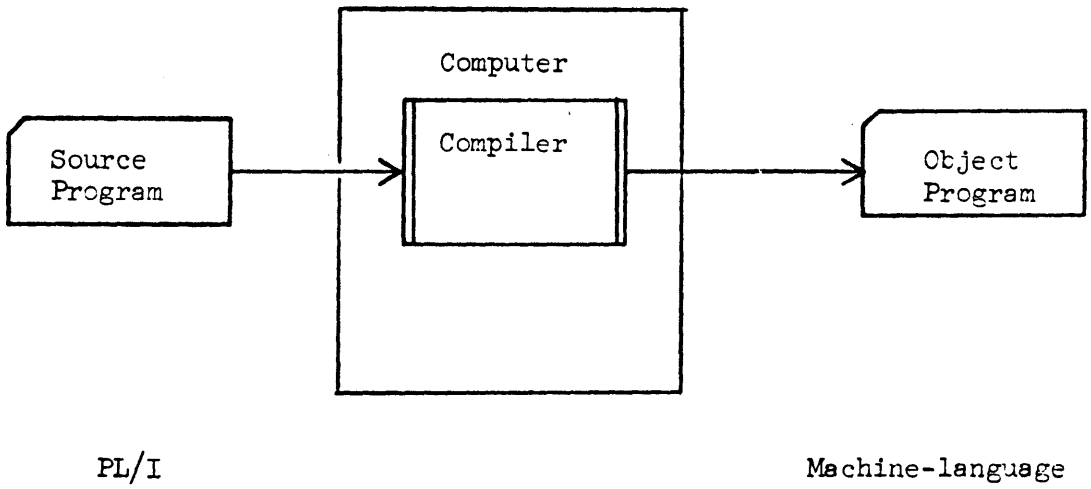


FIGURE 2. A sample CLD page.



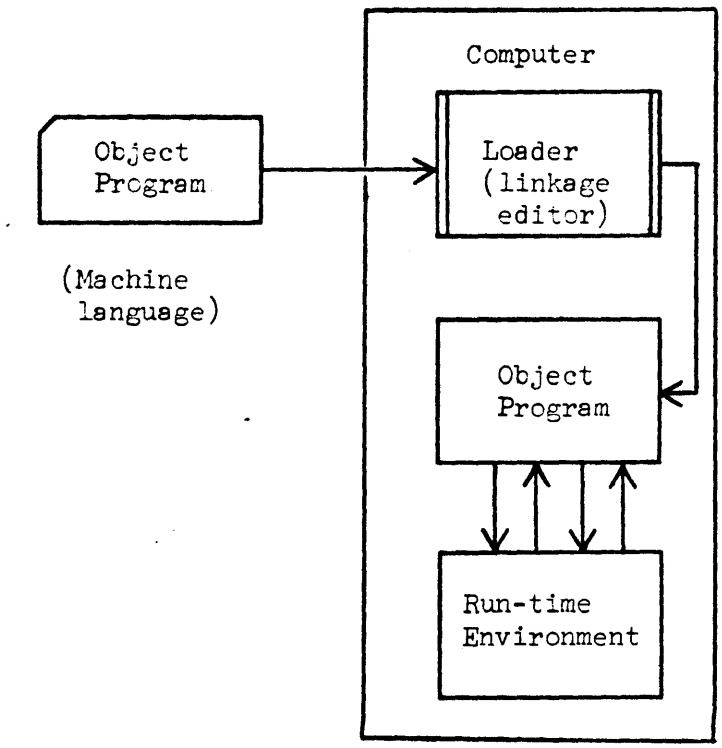
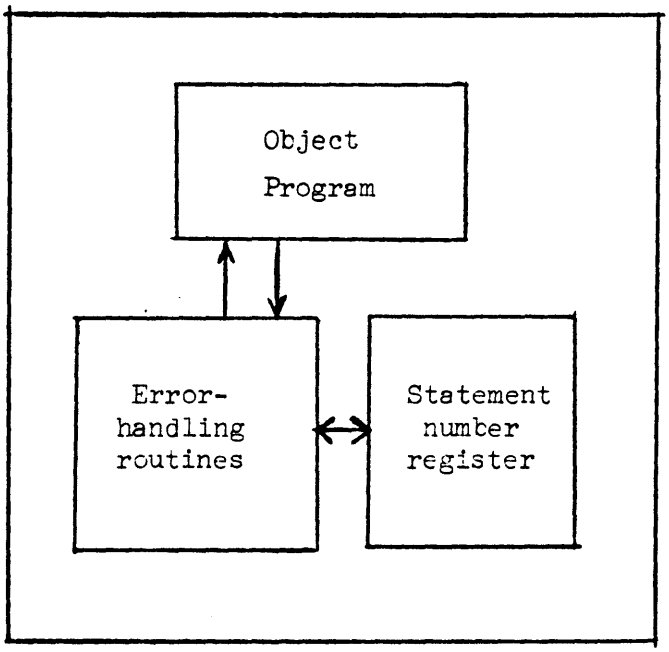




FIGURE 3. Compiling and Loading a Program




Run-time functions

- input/output
- data converstion
- error handling
- debugging aids

FIGURE 4. Run-time Environment.

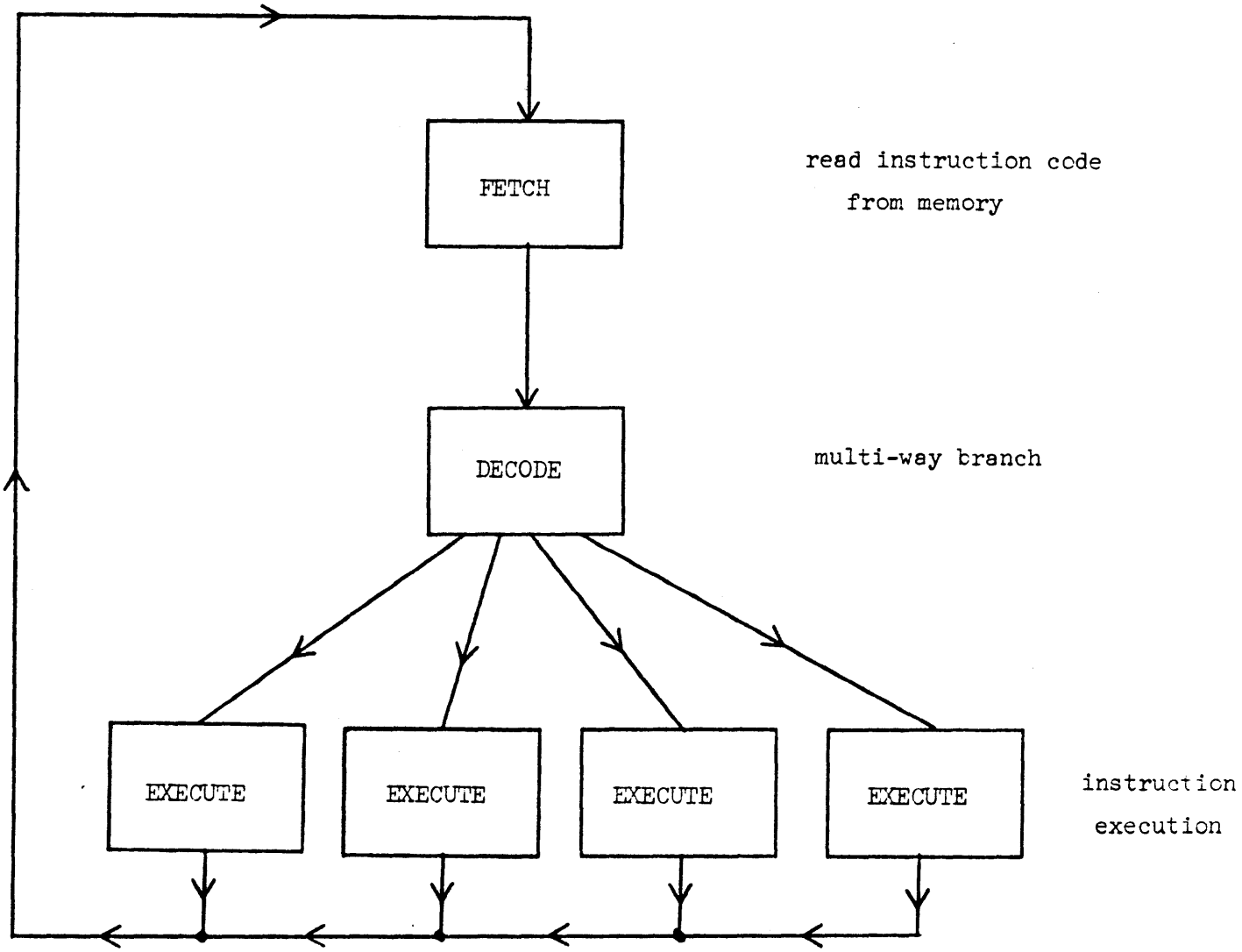
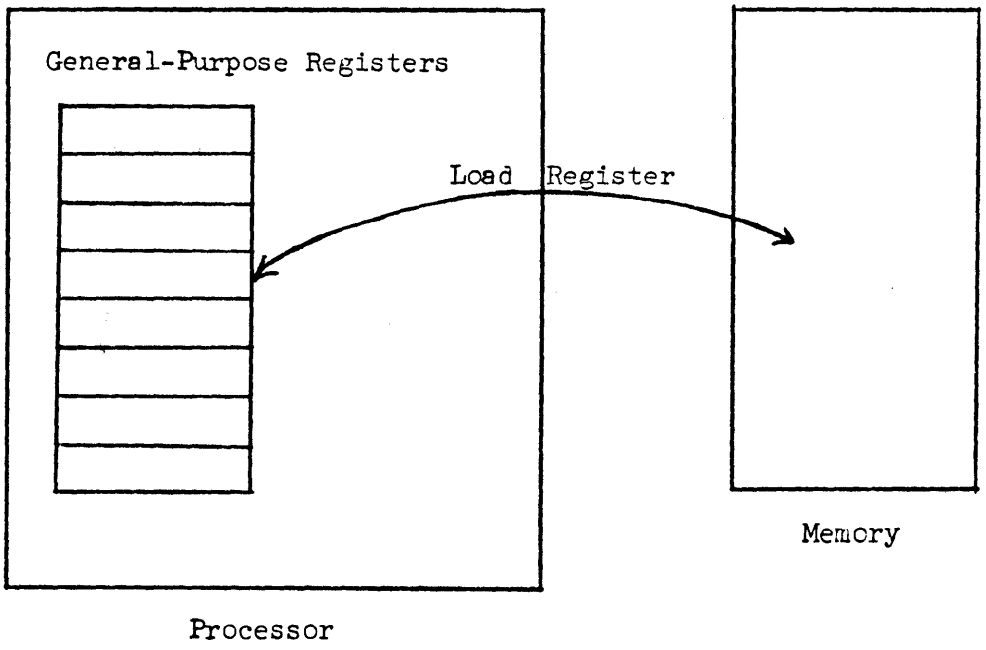
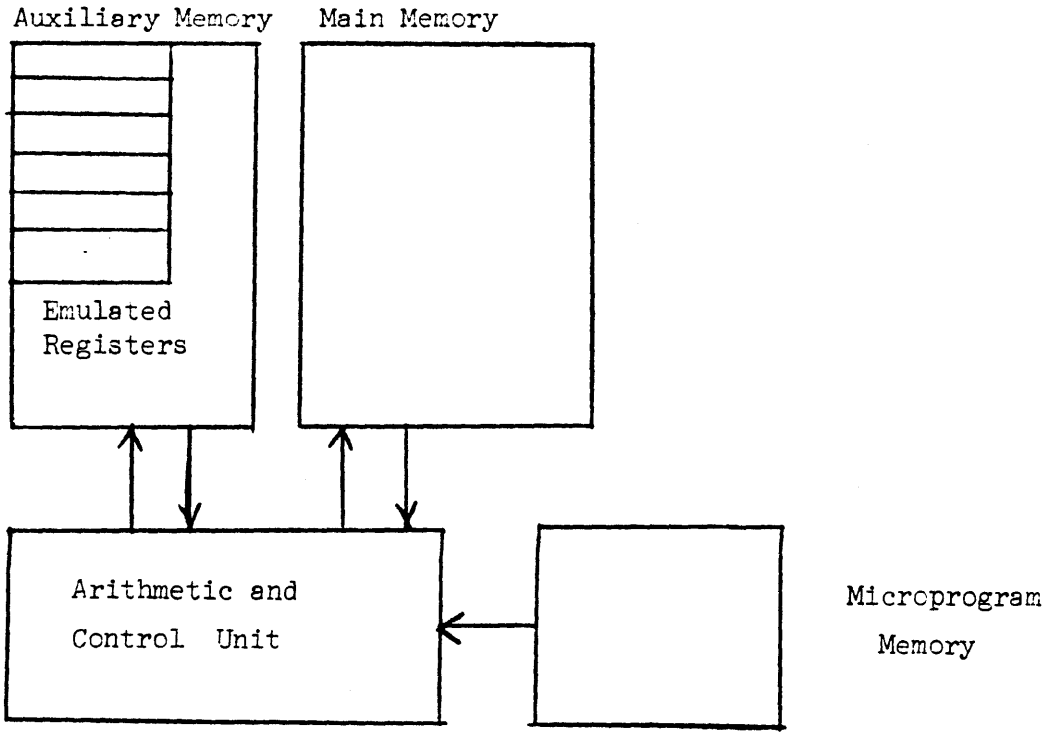



FIGURE 5. Instruction Interpretation Cycle




Architecture




Microprogrammed Organization

FIGURE 6. Architecture versus Organization

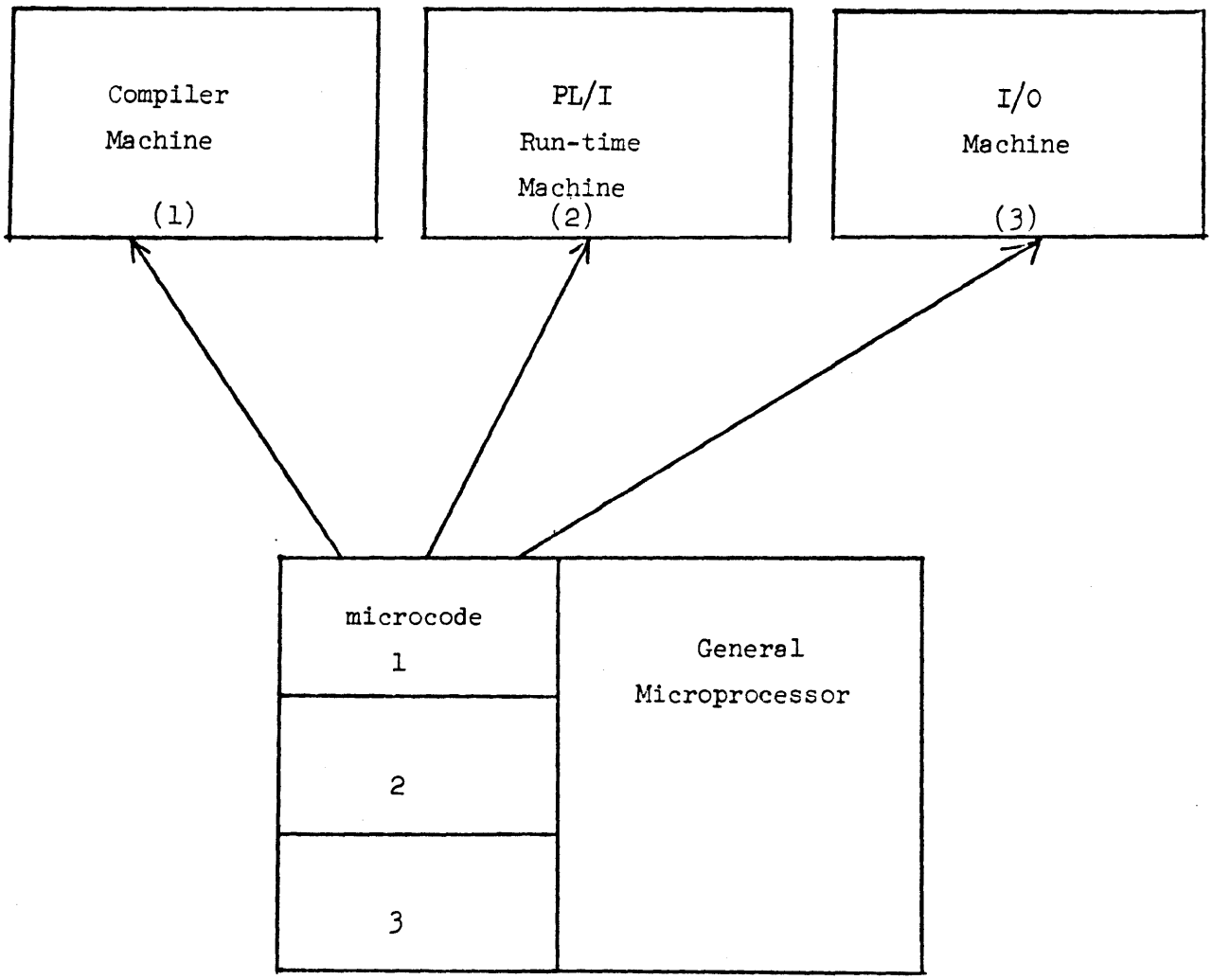



FIGURE 7. Variable Architecture

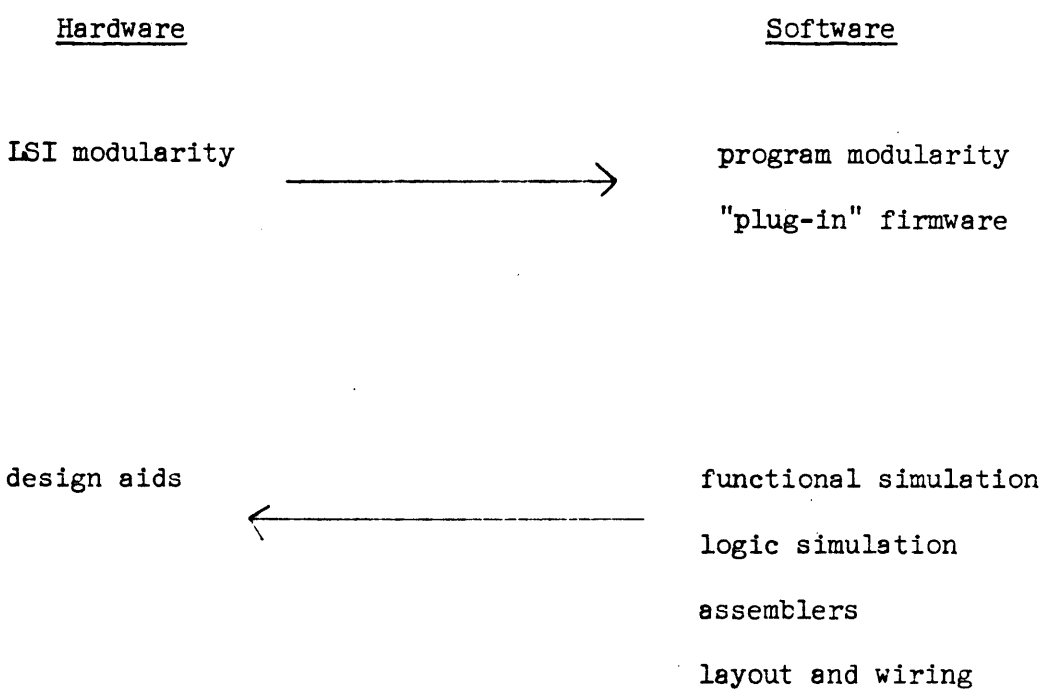



FIGURE 8. Hardware-Software Influences