Rochester Institute of Technology
RIT Digital Institutional Repository

Theses

6-25-1982

On the Use of Cognitive Principles and the Personalized System of Instruction in Computer Science Education

James Bradley

Follow this and additional works at: https://repository.rit.edu/theses

Recommended Citation
Bradley, James, "On the Use of Cognitive Principles and the Personalized System of Instruction in Computer Science Education" (1982). Thesis. Rochester Institute of Technology. Accessed from

This Thesis is brought to you for free and open access by the RIT Libraries. For more information, please contact repository@rit.edu.

Rochester Institute of Technology
School of Computer Science and Technology

On the Use of Cognitive Principles and the Personalized System of Instruction in Computer Science Education

A Thesis submitted in partial fulfillment of
Master of Science in Computer Science Degree Program

By: James Bradley

Approved By: Wiley McKinzie, Advisor

Henry A. Etlinger

Peter Lutz

Date: June 25, 1982

ACKNOWLEDGEMENTS

Above all this, I would like to express my thanks to God for His constant strengthening and encouragement. Without His help, I would not have had the perseverance or energy needed to complete this degree. Also, I would like to thank my wife, Hope, for her patient endurance of my many evenings away from home and my preoccupation with academic matters. I would like to express my heartfelt thanks to my thesis advisor, Wiley McKinzie, for his support, suggestions, and assistance in many ways, and to the others on my committee, Peter Lutz and Henry Etlinger for their generous help. Lastly, I would like to thank the Computer Science faculty at RIT for the many ways they gave of themselves in teaching me and I would like to thank my colleagues at Nazareth College, particularly Herbert Elliott, Sister Dorothea Kunz, and Judith Rose for their many encouragements and patience with me. Without the help and support of all of these, this thesis would not have been possible.

CHAPTER 1

NATURE OF THE PROBLEM

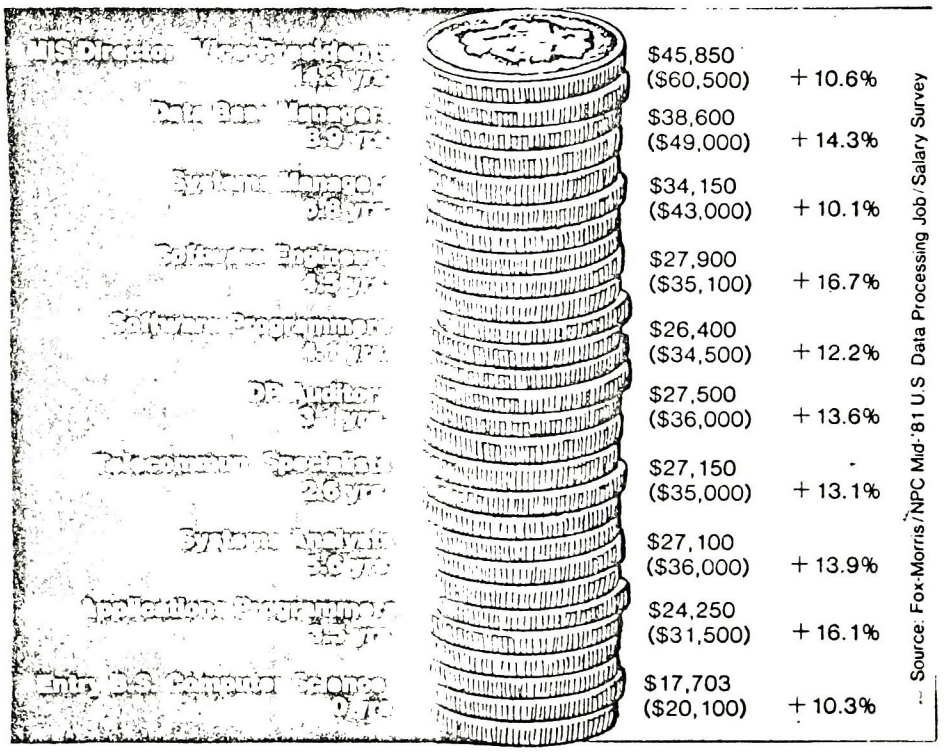
1.1. Introduction

Recently Peter Denning, President of the Association for Computing Machinery, titled his President's letter, "Eating our Seed Corn", (DEN, p.341) referring to the migration of many Computer Science faculty or potential faculty to industry. Many are calling this situation "the crisis in Computer Science Education." The fact that there is a teacher shortage in Computer Science is widely agreed upon. However, there is not widespread agreement on what should be done about it. The remainder of this chapter is a presentation of some statistics regarding the extent of this problem and an analysis of them. Subsequent chapters will deal with one aspect of the solution to this problem -- revisions in the traditional structure of teaching Computer Science courses.

1.2. Demand

The demand for Computer Science professionals is currently very high. A 1981 edition of Computerworld (COM) has noted nationwide survey results which indicate that "computer employees are being sought in 'unprecedented' numbers". The increase in demand was 18.7% in

1980 and was an additional 15.9% through mid-1981. Programmers in particular were in high demand -- an increase of 27.3% occurred between mid-1980 and mid-1981. This demand has pushed salaries up. Computer Science salaries increased 40% in the 3 years 1978 to 1981. Average salaries for Corporate Management System Information Directors were $45,850, for Systems Analysts with 2-5 years experience were $27,100 and for Programmers were $24,250. These increases were tabulated as follows:



The significance of these figures is best seen in comparison to faculty salaries. It is important to note that since Computer Science is a relatively young field, most faculty members within the discipline are in the lower faculty ranks. Thus salaries of the low to mid $20's are

Average Faculty Salaries in 1980-81 and Percentage Increase over 1979-80

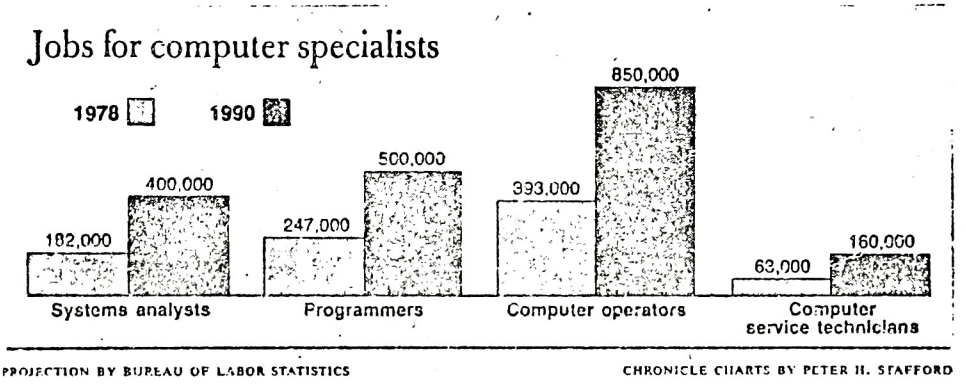
Academic rank	Public		Private independent		Church-related		All combined	
	Average	Increase	Average	Increase	Average	Increase	Average	Increase
Doctorate-granting universities								
Professor	$32,850	9.1%	$36,000	8.6%	$31,170	10.2%	$33,450	9.0%
Associate professor	24,460	8.9%	25,290	9.9%	23,750	9.0%	24,560	9.0%
Assistant professor	19,810	9.2%	20,170	9.6%	19,370	9.1%	19,850	9.2%
Instructor	15,450	10.0%	16,470	11.0%	16,010	10.7%	15,630	10.2%
All ranks	25,730	9.1%	27,930	9.1%	23,880	9.6%	26,060	9.1%
Other institutions offering degrees beyond the baccalaureate								
Professor	$29,580	8.0%	$28,710	8.6%	$24,930	10.2%	$29,000	8.3%
Associate professor	23,440	7.6%	25,560	8.9%	20,530	9.3%	23,000	7.9%
Assistant professor	19,280	7.9%	18,580	9.0%	17,220	8.9%	18,930	8.2%
Instructor	15,500	7.5%	14,810	8.6%	13,700	9.2%	15,190	7.8%
All ranks	23,390	7.8%	22,130	8.8%	20,070	9.5%	22,850	8.1%
Institutions offering only baccalaureate degrees								
Professor	$26,780	8.7%	$27,030	9.4%	$22,540	9.0%	$24,970	9.1%
Associate professor	22,190	8.4%	20,430	9.2%	18,640	8.3%	20,010	8.6%
Assistant professor	18,730	9.0%	16,720	8.9%	15,810	8.4%	16,750	8.7%
Instructor	15,060	8.3%	13,810	9.1%	13,350	8.7%	13,900	8.7%
All ranks	20,740	8.7%	20,210	9.2%	17,890	8.6%	19,300	8.8%
2-year institutions with academic ranks								
Professor	$26,200	8.0%	$23,750	8.2%	$17,720	11.7%	$25,920	8.1%
Associate professor	22,630	6.9%	19,090	7.2%	16,280	8.4%	22,420	7.1%
Assistant professor	19,090	9.2%	14,390	7.8%	14,130	8.8%	18,830	8.6%
Instructor	15,960	7.4%	11,750	7.0%	11,600	9.9%	15,480	7.4%
All ranks	20,740	7.9%	16,010	8.4%	14,440	9.6%	20,390	8.4%
Institutions without academic ranks								
All faculty members	$22,050	9.7%	$15,530	8.2%	$14,710	10.6%	$21,560	9.6%
All institutions except those without academic ranks								
Professor	$31,200	8.7%	$32,650	9.0%	$24,850	9.7%	$30,890	8.6%
Associate professor	23,800	8.2%	23,200	9.7%	20,100	8.8%	23,270	8.5%
Assistant professor	19,470	8.7%	18,660	9.4%	16,750	8.7%	18,970	8.6%
Instructor	15,570	8.3%	14,600	9.2%	13,620	9.1%	15,140	8.6%
All ranks	24,150	8.6%	24,290	9.3%	19,480	9.1%	23,650	8.7%

Note: Average salaries do not include fringe benefits; they are based on data from 2,550 institutions. Percentage increases are based on data from 2,252 institutions reporting comparable salary data for both years.

SOURCE: AMERICAN ASSOCIATION OF UNIVERSITY PROFESSORS

the norm. For instance, according to the above table (CHR2), a college faculty member with 8-10 years experience would typically be an Associate Professor and would typically earn between $20,000 and $25,000. A Data Base Manager or Systems Programming manager with the same experience would earn over $33,000. Part of the reason for this is that salaries for Computer Science faculty are at least partially tied to institutional standards which apply to all departments. Thus even though Computer Scientists may be in much greater demand their salaries

cannot deviate too much from those in other departments where the demand is much less. This high demand for Computer Scientists is seen as persisting past 1990. The following chart was prepared by The Chronicle of Higher Education based on data gathered by The Bureau of Labor Statistics. (CHR1)



These figures represent substantial percentage increases:

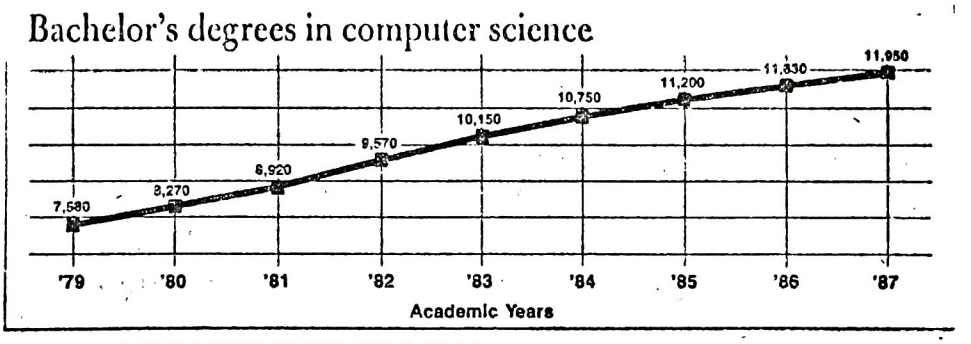
Systems Analysts	120%
Programmers	102%
Computer Operators	116%
Service Technicians	154%

As already cited, the demand for computer professionals increased by 15.9% between mid-1980 and mid-1981. If these growth rates continue, the demand over the 12 year period 1978-1990 will considerably exceed the Bureau of Labor Statistics projections.

1.3. Supply

The supply of computer professionals is not keeping pace with the demand. According to John Hamblen, chairman of the Computer Science Department at the University of Missouri at Rolla, in 1978 American colleges and universities produced only about 1/6 the number of Bachelor's graduates needed, 1/10 the Master's needed and less than 1/3 of the need at the doctoral level. (CHR1)

The National Center for Educational Statistics has produced the following projection of Bachelor's degrees in Computer Science: (CHR1)



PROJECTION BY NATIONAL CENTER FOR EDUCATION STATISTICS

This totals 90,020 graduates through 1987. The Bureau of Labor Statistics projections mentioned above indicate a need for 471,000 additional systems analysts and programmers between 1978 and 1990, positions that normally are filled with people with at least a Bachelor's degree. If the approximately linear growth in graduates foreseen by the NCES is extended through 1990, this would mean a total

of about 140,000 graduates, a shortfall of 331,000.

Thus we have a situation where the demand considerably exceeds the supply and very likely will continue to do so for many years unless dramatic changes in either supply or demand occur. This will continue to push salaries up and will continue to make it attractive for teachers to leave academia for industry exacerbating the problem. It has been suggested that the dynamics of a free market economy will allow the situation to rectify itself. While this may be true in the long run, it will not solve the problem in the next few years. There are at least two reasons for this. One is that there is a delay of four years from the time that someone decides to enter Computer Science and that individual's completion of a Bachelor's degree. Secondly, there are other delays due to the time required for additional schools to add Computer Science departments and for existing schools to expand their programs.

1.4. Students

The NCES statistics cited earlier indicate a 58% increase in the number of Bachelor's degrees awarded in 1987 over 1979. This alone means that there is a considerable need for more faculty and facilities in Computer Science. However, the interest in Computer Science among high school students has increased very dramatically in

recent years. The SAT test includes an optional question regarding anticipated college major. According to Mr. Len Ramist of the Educational Testing Service approximately one million students took the test each year from 1974 through 1981 and about 90% answered this question. The following table presents the percent responding who checked an interest in computer fields:

1974	1 %	1979	3.3%
75	1.6	80	4.2
76	1.9	81	5.2+
77	2.1		
78	2.6		

The 1981 figure had not been finally tabulated as of this writing but Mr. Ramist indicated that it would definitely be in excess of 5.2%. The 1974 figure was only available to the nearest percent.

These figures indicate that the interest in Computer Science among high school students has more than tripled in the seven years from 1975 through 1981. This has produced a number of effects:

-Class sizes in Computer Science in colleges and universities are often very large.

-Many colleges and universities, aware of the anticipated decline in the number of high school graduates, are eager to establish programs in Computer Science.

-Established programs have had to set quotas on the number of students that will be accepted. Thus competition for admission has stiffened.

The result of all this is that the need for Computer Science teachers has increased considerably and will continue to increase in the next few years.

1.5. Faculty

According to Denning (DEN), in 1975 sixty American departments granted Ph.D.'s in Computer Science. By 1979 that figure had increased to 77. However, Ph.D. faculty per department had dropped from 4.3 to 3.2 while faculty per department dropped from 14.6 to 12.4. Ph.D. production has not increased substantially during that time:

Table 1. U.S. Computer Science Ph.D. Production and Faculty (1973–80).

Year	No. Ph.D.s Granted in Computer Science	No. Faculty Holding Ph.D.s	Total No. Faculty
1973	208	†	†
1974	203	787	862
1975	256	805	878
1976	246	773	843
1977	208	790	881
1978	214	825	938
1979	248	837	958
1980	265*	†	†

* Estimate. † Not Available.

This means the total Computer Science faculty in Ph.D. granting institutions increased from 870.0 to 954.4. The

number of faculty holding Ph.D.'s dropped from 258 to 246.4. At this point there is an inconsistency in Denning's data; he claims "a net increase of only 32 Ph.D. faculty during 1975-79". In spite of this inconsistency, however, the trend is clear: relatively little or no increase in faculty accompanied by a substantial increase in undergraduate students. According to Taulbee and Comte (TAU), about 45% of each graduating group of new Ph.D.'s accept academic positions. Between 1975 and 1979 this should mean 528 new faculty. Since Computer Science is such a new field, there is virtually no exodus due to retirement. Thus the lack of growth in faculties can be attributed almost entirely to an exodus of faculty to industry.

Bailes (BAI) has attempted to survey more broadly than just Ph.D. granting institutions. He contacted 463 institutions with Computer Science programs of whom 200 replied. He found the data below on recruitment for full and part time faculty. Also an NSF study identified 600 vacant faculty positions in Computer Science.

Two principal reasons have been identified for this shortage: greater research opportunities in industry and higher salaries. Denning has pointed out that faculty salaries in Computer Science are often unable to deviate from those in other academic departments in which the

Highest Computer Science Degree Offered	FOR THOSE INSTITUTIONS THAT RECRUITED: How many positions?	Ph.D. req.	Filled perm. basis	Filled Temp.	Still open
NONE (33)	9.5	3/10	6 (63%)	1	3.5
ASSOC. (43)	22	1/17	18 (81%)	0	4
B.S. (101)	109	34/71	81 (74%)	2	28
M.S. (57)	86	35/44	60 (69%)	1	26
Ph.D. (50)	109	41/46	80 (73%)	1	29
TOTAL (284)	335.5	114/188	245 (73%)	5	90.5

demand is nowhere near as great. Mitchell (MIT) has pointed out a number of other factors which complicate the situation:

-Retrenchment in many universities means a shortage of money for equipment and salary increases.

-Many secondary and elementary teachers need retooling to handle the growing demand for computer expertise at that level.

-Students in Social Sciences and other areas have a need for training in the use of applications oriented packages.

-A large number of adult learners are seeking computer training. These people are usually job oriented,

require special guidance and counselling and want immediate applicability of what they learn.

The latter three of these factors all add to the demand on Computer Science faculty, while the first makes it considerably more difficult to meet that demand.

1.6. Solutions

Two possible approaches could be used in attempting to solve this problem. One way is to try to increase the number of faculty in Computer Science. Another is to try to increase the productivity of existing faculty.

A number of proposals have been advanced to increase the number of faculty. Some of these are:

-Arranging coalitions between industry and universities so that faculty can earn salaries comparable to their counterparts in industry while possibly serving a consultant role in industry or switching back and forth between the two.

-Treating Computer Science departments like medical schools and exempting them from the usual salary ranges that apply on university campuses. This could require the intervention of business and industrial leaders in attempting to persuade state legislatures to make this possible.

-Having government and/or industry provide substantial funding to regularly upgrade the research capabilities of university Computer Science departments.

-One computer company - Wang Industries - has started its own graduate school of Computer Science. Other companies could follow suit. This is a solution with considerable potential for easing the shortage of computer professionals as computer vendors like Wang are generally in a much stronger financial position right now than schools.

-Give computer personnel who have had years of experience in industry academic standing equivalent to senior faculty with earned doctorates, thus making them eligible for higher salaries and tenure.

While all these proposals have considerable merit, they will not be the focus of this thesis. My focus will be strictly on increasing the productivity of Computer Science faculty. Even if efforts to increase faculty are successful, it will take several years for these efforts to accomplish their work. Thus productivity increases are essential for the immediate future and will prove useful in the long run as well.

Thus Chapter 2 will outline some of the theoretical background necessary for both formulating and evaluating such approaches. Chapter 3 will examine in detail one

major effort at such an improvement. Chapter 4 will conclude by formulating and analyzing some specific proposals.

CHAPTER 2

FOUNDATIONS

One objective of this thesis is to provide some answers to the question, "How can we enable Computer Science faculty to teach more students?". The most simplistic answer would be "Increase class sizes.". However this answer has at least two major flaws: it increases the already heavy workload of the faculty and it may reduce the quality of the education students receive. Hence this approach is inadequate. My contention is that improvements in educational productivity are similar to improvements in computer throughput in that it is very difficult to achieve them without a thorough understanding of how the "system" operates. In other words, it is unwise to try educational innovations without a rigorous foundation in learning theory. With such a foundation, however, (assuming that theory is backed up by solid experimental verification) one has both a measuring stick against which proposed innovations can be measured and a guide for suggesting truly useful innovations.

In seeking such a foundation, this thesis will focus on two men's work: Jean Piaget and Noam Chomsky. Piaget's theories of cognitive development have become widely accepted in recent years and have been (and are being)

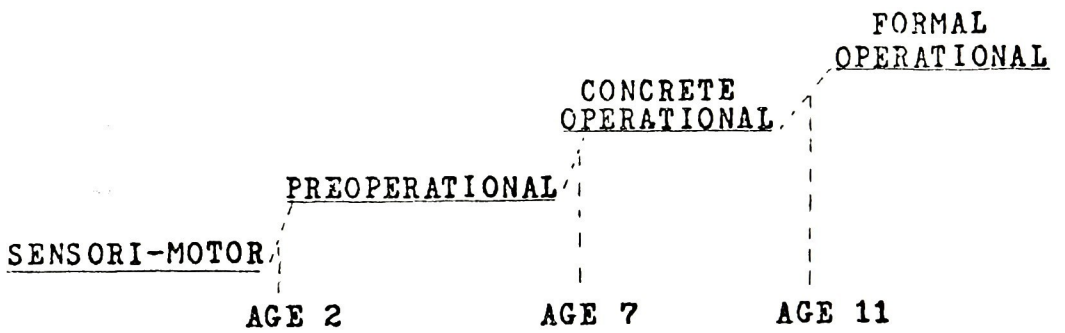
widely tested. Furthermore, there is abundant evidence that his theories have significant implications for education; in fact, they are taught in most teacher training programs in the United States today. Chomsky is a linguist and his theories about natural language have served as the foundation for the development of higher-level computer languages and their compilers. Since Computer Science is a relatively young field, there has been very little study done of how people learn it, unlike older and more established fields like Mathematics or the Natural Sciences. However, there has been a great deal of study done on how people learn natural languages and Chomsky has had a great deal of influence on this field. Thus it seems to me that a great deal could be inferred about the learning of computer language from his work. Of course all such conclusions are tentative and would need experimental verification but that is beyond the scope of this thesis.

In the following sections, then, I will first survey Piaget's theories and their applicability to Computer Science education, then Chomsky's theories and their applicability to Computer Science education, and I will conclude with a summary relating the two.

2.1 Piaget

2.1.1 The Stages of Cognitive Development

From Piaget's perspective, intellectual growth from birth to adulthood proceeds through a series of stages within each of which thinking operates in a relatively consistent fashion, but differently than in the other stages. Thus a crude illustration of Piaget's perspective is given by the following diagram:



This picture is only a first approximation to Piaget's theory in three ways. First of all, the transitional ages vary quite a bit from one individual to another and this variance increases as the higher stages are approached. Secondly, each stage is made up of a number of substages, and, thirdly, a great deal of development takes place within each stage or substage. Also some writers combine the second and third stages into one stage with two substages.

A brief outline of the four periods follows: (FLA)

Sensori-motor period (Ø-2 years):

This stage begins at birth with the child possessing reflexes and undeveloped mental capacities. During this period the capacity to distinguish self from others, to grasp the concept of time, to recognize the permanence of material objects and a great many other intellectual skills are developed. However, there is no internal symbolization of the external reality around the child and no symbolic manipulation of such an inner reality.

Preoperational period (2-7 years):

The principal characteristic of this stage is the development of symbols and the onset of language as one expression of this capacity. The difference between signifier and significate is grasped and "socialization becomes more possible through a system of shared symbols". (FLA) However, the child's perception of the world tends to be kaleidoscopic in that he or she grasps one image after another rather than an overall perspective on what is happening.

Concrete Operational Stage (7-11 years):

This stage is characterized by a conceptual organization of the world around. The child is now able not only

to symbolize but to perform operations on these symbols. The kind of operations Piaget has in mind include, for example, the capacity to group symbols together and form classes, to add classes together and to subtract classes. In mathematical terms, one could say that the child has intuitively grasped the concept of set, can form sets, and can perform the set operations of union, complement, and difference. With this capability also comes the capacity to deal with relations and perform relational operations. For example, a child this age uses the transitive law -- $A < B$ and $B < C$ implies $A < C$. Arithmetic soon follows. A classic example of the difference between the pre-operational and the concrete operational child is Piaget's conservation experiment. A tall thin container is filled with water and the water is poured into a shallow flat dish. A child is then asked whether there was more water before or after the pouring. The preoperational child picks one configuration or the other and announces that that one has more. Even pouring the water back into the original container does not convince him or her that neither has more. The concrete operational child realizes that neither configuration has more even without the demonstration. Piaget explains that the older child is mentally able to perform the operation of restoring the water back to the original container (reversing the process) realizing thereby that the amount of water is the

same both ways. I.e., the younger child is unable to perform and unperform the operation mentally without being caught up in a single concrete reality perceived in isolation. This kind of thinking is the beginning of being able to extend from the actual toward the potential.

Formal Operational (11+ years):

Piaget identifies three principal intellectual capacities which first appear in this stage. The first is true hypothetico-deductive reasoning, i.e., reasoning in which possibilities for reality can exist each of which can be examined and investigated for correspondence with one's own experience. Secondly, abstract propositional reasonings such as the classical syllogism:

if a implies b
and a is true
then b is true

are not only meaningful but become a natural part of thought. Flavell has expressed this distinction by saying that "concrete operations are intrapropositional, since they go to make up the content of individual propositions, whereas formal operations are interpropositional, since they involve logical operations among the propositions thus formed." (FLA, p.206) A third key characteristic of this stage is combinatorial analysis, that is, a systematic isolation of all the variables in a situation plus

all combinations of these variables. These can then be systematically evaluated. Flavell summarizes this stage of development like this:

> "..the adolescent..begins by organizing the various elements of the raw data with the concrete-operational techniques of middle childhood. These elements are then cast in the form of statements or propositions which can be combined in various ways. Through the method of combinatorial analysis he then isolates for consideration the totality of distinct combinations of these propositions. These combinations are regarded as hypotheses, some of which will be confirmed and some infirmed by subsequent investigation. Is it true that A elicits X? If so, does B also? Is it true that A produces X only when B is absent? Such are the hypothetical questions which make up the domain of the possible in such problems; and the adolescent views his task as that of determining the actual shape of things by successively putting them to empirical test."

In other words, by the concrete operational stage it is possible for a child to form internal symbols of external realities and to manipulate these symbols instead of having to manipulate the physical objects. In the formal operational stage, the individual is able to form symbols of groups of symbols and of abstract ideas. Hence he or she can manipulate not only symbols but the logical patterns by which symbols (and hence ideas) relate to each other.

2.1.2 The Growth Process

In Piaget's viewpoint, moving from one stage to another or even moving within a stage proceeds via "assimilation" and "accomodation" - two concepts drawn from Biology. Assimilation means absorbing an experience, event, or concept into the existing framework of thought one already has. Accomodation means the changing of the individual's own thought system to adjust for the new experience that has been presented. The process is analogous to bootstrapping a computer in which successively more complex systems are loaded in with the help of earlier ones. The child starts with almost nothing (in the way of concepts, that is) and gradually adds new understandings. However, eventually the system that develops reaches its capacity and the impact of further stimuli which cannot be integrated into it cause a radical accomodation and the organization of a new system at a higher level. It is important to note that in both assimilation and accomodation, it is the perception of conflict with one's existing way of thinking that leads to growth. If the new data can be integrated into the existing system, they are. If not, the existing system is reorganized (sometimes a little, sometimes a lot) to handle it.

Piaget views these stages as being genetically determined, hence the name he gives to his studies, "Genetic Epistemology". There has been some effort to verify this view by cross-cultural studies. Evidence tentatively

suggests that that the same stages are proceeded through in the same order independently of culture. However, people in some cultures are more likely to reach the formal operational stage than in others. Also, tremendous variation has been found in the age at which people reach this level. One summary notes that the percentage of college students who use formal logic when attempting to solve problems varies from a low of 11% to a high of 61% in different studies. (HAL, p.407)

2.1.3 Implications for Computer Science Education

One principal conclusion I want to draw from Piaget's work is that there is such a thing as "programming readiness". An individual is "ready" to learn to program when he or she is a formal operational thinker. First I will examine some of the evidence for this conclusion then I will draw some further implications from it.

Let us compare each of the three main characteristics of formal operational thought discussed earlier with the process of writing a computer program. Combinatorial analysis is essential in thinking through all the possible data configurations which can occur, in planning one's logic to allow for them, and in the preparation of test data. The capacity to formulate hypotheses, work out

their implications, and compare them to reality is crucial in designing a program to accomplish a particular task; one must consider all possible data inputs the program could receive and all possible outputs. Propositional thinking is used in setting up conditionals, loops, and the tests these use. Thus, in its very essence, computer programming requires as a prerequisite that an individual be a formal operational thinker. One would expect that individuals who are consistently formal operational thinkers would be able to grasp programming fairly easily and that individuals who are not would probably not be able to do it. Also people who are in the transitional stage between the concrete and formal levels would be inconsistent in their programming performance since they are inconsistent in their use of formal thought.

There is also experimental evidence that formal thought is a prerequisite for programming. Unfortunately, there has apparently been only one study done investigating the implications of Piaget's work for Computer Science Education and that used such a small sample size that its results must be regarded as tentative. In this study (KUR) the author prepared a fifteen item test of formal operational reasoning. (This test is in Appendix 1.) It was given to a class of 23 students in an introductory programming course. Based on their scores the author classified their cognitive level as late concrete, early

formal (i.e., transitional), or late formal. Their performance in the course was as follows:

	low (C-,D)	average (B,B-,C+)	high (A,A-)	total
late concrete	3	Ø	Ø	3
early formal	4	7	2	13
late formal	Ø	Ø	7	7
total	7	7	9	23

In this class all the formal thinkers scored very well, all the concrete thinkers did very poorly, and the ones in transition were inconsistent, exactly what the theory would lead us to expect. Analysis of variance of the results showed that the characteristics of sex, class level, and number of previous Math-Science courses were able to explain less than 5% of the variation in the grade. However, cognitive level explained 62.4% of the variance. In fact, it explained 80% of the variation in test scores and 39% of the variation in programs. This was attributed to the availability of help in programs.

Although there is only this one study showing the correlation of formal operational thought and success in programming, there is abundant evidence from other discip-

lines that formal operational thought is a key to success in situations that involve logical reasoning and problem solving. Thus I feel three conclusions can be drawn:

-There is a need for further testing of the hypothesis that formal operational thought is the principal prerequisite for computer programming.

-Even while awaiting further experimental results, I suggest that the evidence is sufficiently strong and the need sufficiently great that we can start now to use tests for formal operational thought as measures of programming readiness. The data that are gathered through this process could be used to provide some of these further results that are needed. Kurtz feels, however, that his test needs to be shortened somewhat in order that students can complete it in 50 minutes.

-Students who are not formal thinkers should be encouraged to enroll in a course designed to develop formal reasoning skills. Courses such as this are fairly common today and models are readily available. (COL) It is possible that in the future students who are not formal thinkers could be required to take such a course before programming; however, I do not feel the evidence is strong enough yet to warrant requiring it. It is conceivable that some beginning programming could be a part of such a course. However, the learning of reasoning skills, not

programming skills should be the principal objective of the course. This one single step would relieve Computer Science teachers of a good deal of time spent (and probably wasted) helping students who are really not ready for programming.

There are some further implications of Piaget's work specifically for the teaching of programming. Assuming we are teaching only formal operational thinkers, learning programming for these people should be fairly easy. Nothing intrinsically new in terms of logical reasoning, systematic evaluation of possibilities, and logical structures is being presented to them although these are being seen in a new context. Learning should proceed more by assimilation than accomodation; the accomodations that need to be made do not require major cognitive reorganization. Good, Mellon, and Krumhaut (GMK) have identified six appropriate teaching techniques based on a Piagetian approach. All of these are applicable to Computer Science.

-Beginning with familiar, concrete examples.

-Emphasizing a "noncookbook" laboratory approach to identifying and solving problems.

-Questioning all answers.

-Intentionally introducing (apparent) contradictions that cause cognitive conflict.

-Encouraging student interaction in considering ideas, problems, etc.

-Removing obstacles to maximum exploration of ideas.

2.2 Chomsky

2.2.1 Three Approaches to the Teaching of Natural Language

The traditional approach to the teaching of natural language is the grammar-translation approach. Most of us who studied foreign language before the mid-60's were taught by this method. Based on the use of prescriptive rules and explanations in the language of the learner (CHA, p.58-60), it grew out of the classic education methods used to teach Latin and Greek. Its purpose was to prepare an individual for the study of literature, to develop an understanding of his or her native language, and to develop the ability to learn.

However, due largely to the influence of anthropologists who stressed the oral nature of language and psychologists from the stimulus-response school of B.F. Skinner, the audio-lingual approach was developed in the late 50's and early 60's. The principal idea of this school is that language is a learned response. Hence students should learn by responding to prompts and being rewarded for correct responses. The system should be so structured that the opportunity for incorrect responses is small. An emphasis was placed on rote learning and

drills. "Overlearning", learning a language pattern until it becomes an automatic, non-thoughtful response, was valued. Emphasis was primarily on structure rather that on vocabulary. Units in a course were organized around a dialogue in the language to be learned. The student memorized this dialogue, then participated in pattern practice exercises to master the structures of the language presented in the dialogue. These were followed by application activities. Teaching was done (ideally) without the use of the natural language; students were to learn grammar inductively -- through repetition of the pattern drills. (CHA)

A number of complaints about this system soon surfaced. The bilingualism it promised had not materialized. The repetition required for overlearning was boring. Avoiding grammar discussions until structures were overlearned was time-consuming and frustrating. Also, teachers found they could not eliminate English completely from the classroom. But the major blow to audio-lingual theory was a book review of B.F. Skinner's Verbal Behavior written by Noam Chomsky. (CHO) Skinner's book was the result of over 20 years of research on verbal behavior from a behavioristic perspective. The concepts that it presents were the foundation for the audio-lingual theory. Chomsky demonstrated two things: the concepts of behavioral psychology as Skinner applied them to verbal behavior

become vague and virtually contentless when carefully analyzed and language is incredibly more complicated than Skinner's theory allows. In fact, language use is infinitely varied. As Chastain puts it, "No one learns English by learning any particular sentences of English." Also Chomsky asserts the existence of "deep structures" in language -- structures that are held unconsciously and that are impossible to account for within the context of behavioral thought.

The outcome of Chomsky's linguistic work and studies done by others on the audio-lingual approach was that both its theoretical foundation and its methodology came to be seen as inadequate. The preferred view today is the cognitive approach. Its insights are drawn from cognitive psychology (which includes Piaget) and from Chomsky's generative-transformational school of linguistics. The basic idea is that individuals proceed from mental awareness and understanding to practice. (In Chomsky's terminology, "Competence precedes performance.") The approach is deductive, not inductive and the emphasis is on conceptually grasping the structures of the language being taught first, seeing it used in context, then moving out toward expression. In this sense it is more like the grammar-translation approach than the audio-lingual approach. However, grammatical terminology is not used -- emphasis is on comprehending a structure for use, not on

developing a systematic grammar along the lines of Latin or Greek grammar. Comprehension is seen as coming from contrasting comparisons with the native language. Also making mistakes is quite acceptable. Correction of such mistakes is seen as one of the major instruments of learning as what is learned is not a system of automatic responses but a cognitive structure or framework which is then used to generate correct sentences in the target language.

2.2.2 Some Key Concepts in Chomsky's Work

I have already mentioned the ideas of competence and performance. Competence means possessing the internal cognitive structure for the language. From Chomsky's perspective, the use of language is governed by rules. These rules are internalized in terms of the deep structure of a language, as contrasted to the "surface structure" -- what one reads, hears, or says when communicating in a given language. It is basically the same as the differences between the generative rules for the grammar of a computer language and the statements written in that language. Chomsky believes that every individual, in learning a language, develops a set of such generative rules and that these are used to produce utterances in the language. This is why individuals are capable of both producing and comprehending statements they have never encountered

before, a phenomenon that stimulus-response theory cannot account for. Furthermore, there exist transformational rules which allow an individual to take apparently different sentences (different surface structures) yet perceive that they have the same meaning. In Chomsky's view, the capacity of human beings to develop such a grammatical structure is genetic. The existence of deep structure is an aspect of universal grammar -- the common feature of all languages which Chomsky believes exists and is intrinsic to the human brain. He views it as a biological rather than a psychological entity, that is, more like an organ of the body rather than something which is learned. Thus for Chomsky, language learning is a process of discovering which of many possible grammars an individual is being exposed to. The structure of this grammar is already embedded in the intellect.

2.2.3 Applications of Chomsky's Work

Learning to program involves the acquisition of problem solving and reasoning skills as pointed out earlier, but it also involves the acquisition of the syntax of a particular language. Piaget's work has tremendous implications for learning the problem solving aspects of programming; I would also like to suggest that Chomsky's work has a lot to say about the learning of syntax.

Although I have only been able to find one study applying Piaget's work to Computer Science education, there is abundant evidence from many studies done by psychologists and educators in other disciplines to give credence to the implications of his work. However, at this time, there has apparently been nothing done to draw out implications of Chomsky's work for Computer Science education and test them in any systematic way. Thus whatever conclusions are suggested here are necessarily very tentative although not unreasonable since his approach has been successful in natural language learning and since his concepts have been the basis of much development in computer languages. In attempting to spell out some implications of Chomsky's work for Computer Science, I will make two major assumptions, the first somewhat verified, the second not verified at all. The first assumption is that Chomsky's view of language as being generated from deep structure is correct; the second is that when a computer language is learned (not only a natural language!) a deep structure of the same type is acquired, although it is quite a bit simpler. In other words, I assume that in learning a computer language, an individual is unconsciously structuring a set of generative rules for that language within his or her own intellect. From these assumptions a number of conclusions follow. Many of these suggestions are adaptations of similar suggestions by

Chastain for the teaching of natural language.

-The key to learning a computer language is learning deep structure not surface structure. Points of grammar and syntax find their principal meaning in the contribution they make to clarifying the deep structure of the language not as ends in themselves. Thus the teacher needs to focus on bringing students to an understanding of underlying patterns not syntax details.

-The explicit presentation of the fundamental patterns of the language and identifying them as such should enhance the acquisition of deep structure. The use of syntax diagrams such as are commonly used in Pascal should aid in this.

-It would be very helpful for the teacher of a computer language to understand how its compiler works or at least understand the generative rules on which the language is based.

-Analogies to natural language for learners of a first computer language would be very helpful. Learning of further languages would be enhanced by frequent comparisons to languages already learned.

-Transformational rules as well as generative rules should be taught. For example, a student should not only be taught to express a particular algorithm using a

particular kind of loop, but at some point should learn to express that same algorithm using different loops. These different kinds of loops should then be compared to see when each is more suitable.

-Building on prerequisite knowledge and the use of explanation (a deductive rather than an inductive or discovery approach) is very helpful.

-Interaction with a native speaker is very important in natural language acquisition as the trial and error experience of communication enables the formation of the correct deep structure. With a computer language, the native speaker is the computer. Thus considerable programming experience is important. Also the use of creative programs that cannot be written by routinely mimicking textbook or classroom examples is extremely important.

-Comprehension of errors is more important than repetition of correct patterns. Thus having help available in the lab and providing comments on corrected programs is very important.

-Use of graphics and schematics is helpful.

-Exercises that require choice and decision are better than drill.

-Organization of a course should be around key aspects of the deep structure that need to be acquired.

-Requiring students to compare and articulate language rules is a major factor in reducing errors. Also students should verbalize principles while solving problems and make generalizations when they are done.

It is worth noting here that there is no evidence that the now-traditional language lab used in foreign language learning has been any help (KBW). Students using a language lab do no better than control groups taught without it. This may be due to the fact that it was organized upon principles drawn from stimulus-response theory. Thus the traditional foreign language lab should not serve as a model for computer labs.

2.3 Summary

The principal similarity between these two men's work is their emphasis on cognitive structure. The principal difference between them is where that structure comes from. For Piaget, the capacity for cognitive structures is genetic, but the structures themselves (that he looks at) develop through interaction with the environment. For Chomsky, the structures of language are themselves genetic. In terms of the implications for Computer Science, however, this difference is immaterial.

Another major similarity that has significance for Computer Science education is that learning programming is a process of developing intellectual structures rather than the accumulation of facts, skills, or learned responses. Other major points of similarity are that for both learning comes through meaningful interaction. Piaget is more overt in pointing out that learning comes through conflict between the inner model of reality one has constructed so far and the external reality itself. However, Chomsky suggests the same idea in saying that language learning comes through the making of errors and the correction of those errors.

One last conclusion that can be drawn is very important, although I have to emphasize that it needs experimental verification. Piaget's work suggests the conclusion that persons need to be formal operational thinkers before learning programming. This means they need to be at least 11 years old and for many people even older. Chomsky's work suggests that once an individual reaches puberty, the capacity to acquire the deep structures of a language decline. Thus the best time to teach programming would be as soon as possible after the onset of formal reasoning. This varies a great deal among individuals. Thus it is impossible to set a single grade level as the ideal place to introduce programming. However, for many individuals, Junior High would be the appropriate level.

This also suggests that elementary school students could benefit a great deal from concrete experience using computers (this is what Seymour Pappert is attempting to do with his Logo language) and could conceivably benefit from learning syntax. But attempts to teach programming at this age would be unwise.

CHAPTER 3

SOLUTIONS CURRENTLY BEING TRIED

In the SIGCSE literature, there are a number of papers detailing approaches that authors have found successful in improving effectiveness and/or efficiency of various courses. However, the work of Kenneth Bowles and his associates in developing the UCSD Pascal system is far more extensive and thorough than anything else being done today along these lines. As part of this, he has developed an introductory programming course which can serve as one possible model for improving productivity in Computer Science Education. This chapter will consist of a discussion of Bowles' work and of the concept of PSI (Personalized System of Instruction), followed by a brief section discussing another tool often proposed for improving educational productivity - Computer Assisted Instruction.

One of the major aspects of Bowles' work is the UCSD Pascal system itself. This is a version of Pascal which includes almost all of the features of the language as Wirth designed it plus a number of enhancements. It is written for use on microprocessors although it can also be run on a number of minis and is relatively machine independent. Although this system has a great deal of interest in itself, I will not discuss it in any detail;

my main interest here is his course. Thus Section 1 will deal with teaching programming as Bowles has organized it. Specifically this section consists of a general presentation of PSI and its philosophy and how it is implemented by Bowles, a discussion of its strengths and weaknesses, and a discussion of some possible enhancements that could be made to it.

3.1 The Introductory Programming Course

3.1.1 PSI

The Personalized System of Instruction was developed in 1962 when Fred Keller , one other North American, and two Brazilian psychologists were asked to form a Dept. of Psychology at the University of Brasilia. They attempted to develop a teaching method based on their own theoretical persuasion, reinforcement theory, which is closely related to the stimulus-response theory of learning associated with B. F. Skinner. The Brasilian venture came to an abrupt end due to political upheavels in that country. However, enough experience had already been gathered with PSI that Keller and his colleagues became quite enthusiastic about it.

Keller himself has summarized what he regards as the principal features of PSI: (KEL)

-The go-at-your-own pace feature, which permits a

student to work through the course at a speed commensurate with his ability and other demands upon his time.

-The unit perfection requirement for advance, which lets the student go ahead to new material only after demonstrating mastery of that which preceded.

-The use of lectures and demonstrations as vehicles of motivation, rather than sources of critical information.

-The related stress upon the written word in teacher-student communication; and finally:

-The use of proctors which permits repeated testing, immediate scoring, almost unavoidable tutoring, and a marked enhancement of the personal-social aspect of the educational process.

Looking at it in terms of reinforcement theory, the idea is that the student will learn the "appropriate responses" if he or she "gets reinforced". The appropriate responses here are answers to factual questions or demonstrations of understanding at a deeper level through explanation of a concept or process to a proctor. The reinforcement is personal attention from the proctor, being told that one is correct (or incorrect) immediately after testing (and testing is quite frequent), and the satisfaction that comes from moving steadily along towards a goal. There are several differences between this and the traditional lecture approach. Lectures, as Keller originally conceived PSI, were totally optional and were to be used as a motivation. That is, when enough students had reached a certain point in the course, a lecture on that topic would

be announced. Only students who had progressed that far were allowed to come. Mastery of a unit was required before a student could go on. The principal interaction the student had was with his or her proctor rather than the professor. Material was acquired primarily through reading, rather than through oral presentation in a lecture.

3.1.2 How PSI is Implemented by Bowles

PSI has been modified in many ways since Keller's original formulation. For example, lectures, as Keller suggested using them, have not been found to be an effective motivational tool and, in many applications of PSI, have been dropped.

Bowles has retained some of these changes and added some more. Part of his own description of the course follows: (EDU, p.22)

> Students in our course are given a brief orientation to the class organization, and to hands-on use of the microcomputers, during the first week of the academic quarter. Thereafter, they proceed through the course materials in the textbook and study guide at their individual paces. The organized class activity is concentrated in a teaching laboratory containing the microcomputers, which is open at least 80 hours per week. At all hours when the laboratory is open, it is staffed by two or more "proctors" who function as learning assistants. Duties of the proctors include grading of homework and quizzes, and providing consulting assistance to students. The proctors are usually experienced students, either graduate or undergraduate. Quite often, the best proctors are under-

> graduate students who scored well in the same course in a previous term.

The course is divided into units and each unit has both homework and a quiz associated with it. The concept of mastery learning is retained from Keller's formulation. One problem Bowles noted with PSI is that students will often choose a proctor who grades less conscientiously than the average. Bowles has attempted to correct this by automating quizzes. The quizzes use random selection so as to vary them quite extensively. Also the computer responds to incorrect answers by displaying the correct answer and a brief explanation.

As a result of these additions, Bowles feels his PSI course has been improved in several ways: (EDU, p.23)

> First, it has become practical to require each student to take a quiz covering all of the topics in a study unit. We thus avoid the quiz-selection tactics of the lazy students. Secondly, it has become possible to assure that all proctors have proficiency in all of the subject material of the course. We have found that even the most experienced graduate-student proctors have identified as "bugs" in the quiz programs, detailed properties of the course material that are in fact correct. Third, the time taken by the proctors to accomplish record-keeping tasks has been virtually eliminated through automated recording of the student's performance on formal quizzes. Fourth, we require each individual student to submit a few complete computer programs for automatic grading by a testing program. It has proven extremely difficult for the proctors to review programs written by the students for homework in such a way as to assure correct results or full understanding by the individual student. The testing program now

> assures that each student emerges from the course with at least the minimum desired capability to solve problems on the computer. In combination, these factors have doubled or tripled the amount of proctor time available for direct consultation with students on the more subtle and less routine aspects of the course.

Bowles emphasizes non-numerical problem-solving to make the material as widely accessible as possible. Also he is attempting to organize the course in such a way as to make it easy for students to learn via self-study. Lectures are used at the beginning of the course to orient students to PSI. After this lecture attendance drops off rapidly and Bowles typically discontinues them after about the third week.

3.1.3 Critique

PSI is purely a method of organizing a course. It says nothing about the content of what is learned. However, abundant research has shown it to be successful as a teaching tool. Specifically, students who complete courses taught by PSI generally average significantly better on tests than students who study the same material taught by lecture.

The fact that PSI is successful poses a significant theoretical problem. PSI clearly has its roots in Skinnerian behaviorism which has been shown to be ineffective as a basis for teaching natural language. Yet PSI has

been shown to be an effective teaching tool in many different fields. The resolution of this "paradox" is not difficult. The reason that behaviorism failed in natural language teaching was that it did not recognize the existence of "deep structure". It was based on the assumption that language is a highly complex skill made up of many little skills (and Chomsky showed language learning is not primarily skill acquisition). For Skinner, language is a "big skill" which can be taught by frequent reinforcement of the correct performance of little skills. This is the same process that Skinner used to train pigeons to perform complex movements and he felt that human beings could be taught language in the same way. In fact, he denied the existence of deep structure.

PSI makes no such claim as to what it means to know something. It should rightly be seen as an administrative tool; it is simply an approach to organizing a course -- it says nothing about the content of that course. The fundamental behaviorist concept -- breaking complex skills or knowledge down into a collection of small steps, structuring a situation in which individuals will perform those small steps correctly, and providing reinforcement for correct performance -- is totally lacking in PSI. Thus PSI is not behaviorism, in spite of its roots. In fact, it is assumed the student will make mistakes and one of the purposes of the grading interview is to correct those

mistakes. Such an approach is antithetical to behaviorism; it implicitly assumes the existence of cognitive structures which can be changed by simple correction rather than by the complex retraining process that behaviorism would see as necessary.

There are several aspects of PSI that are quite consistent with the kind of approaches that both Piaget's and Chomsky's theories suggest. (See Sections 2.1.3 and 2.2.3). The personal interaction with a proctor provides stimulus, challenge, and above all, correction where the student's concepts are incorrect. Bowles himself points out that the grading interview is generally regarded as the principal source of the success of PSI. He feels that this is where most real "teaching" takes place in his course. (EDU, p.22) It is especially important to note here that the "key" to PSI's success is direct, personal interaction with another individual. This is precisely the same approach to learning that both Piaget's and Chomsky's theories lead to. Thus, although the theoretical roots of PSI are antithetical to cognitive theory, the conclusions both draw about how to teach are the same.

PSI is consistent with the recommendations of cognitive theories in other ways as well. Frequent quizzes provide for more interaction and correction than the relatively infrequent quizzes of a typical lecture course.

The student must take responsibility for his or her own learning. Mastery is required. Thus the student cannot simply skip areas that seem difficult or unappealing counting on his or her knowledge of other areas to earn a passing grade or simply hoping that questions on those areas will not be asked. The challenge and confrontation with another individual that take place in the grading interview is exactly what Piaget's theories suggest as a teaching tool. Also, there is nothing in PSI that inhibits the preparation of materials and assignments in such a way that they emphasize a non-cookbook approach, stress real problem-solving, and stimulate thought.

PSI has a number of weaknesses, however. First of all, it has a much higher attrition rate than lecture-based courses and it yields many more incompletes. It is not hard to see why this happens. Not everyone is sufficiently motivated by attention, approval, and success to enable them to maintain the pace necessary to complete a course. Another way of looking at it is that many students lack the self-discipline necessary to complete a self-paced course, since the threat of failure if they do not keep up with the class pace is gone. Secondly, persons who tend to be individual learners do well with PSI, but ones who depend on interaction with peers for their learning do not do so well.

Along the same lines, one of the main criticisms of PSI is that it does not provide a role model for students nor does it provide for socialization. This criticism is particularly significant when applied to the first course in Computer Science. It is in this course that the concept of Structured Programming is introduced. Computer Science departments typically try very hard to transmit to students a set of professional standards (or values) which are expressed by the concept of Structured Programming. It is widely felt that it is important for students to get into the habit of writing structured programs right from the beginning. It is possible that committment to such a value cannot be communicated as well through the reading and grading interviews of PSI as through a lecture-discussion approach. However, this conjecture needs research.

A different kind of weakness was pointed out by one of my students who commented, "You can't get as much in an hour's reading as a teacher can tell you in an hour." This student and other students have expressed a negative attitude toward PSI because they perceive it as requiring quite a bit more work than a lecture course. Also PSI as presently done is not suitable for students who do not read well. Colleges are seeing increasing numbers of these students. Another problem in PSI courses is lab exercises. Students who complete the labs early could

make their corrected programs available to slower students. The security here is more difficult than with quizzes as it is considerably more difficult to individualize lab assignments and have them be approximately equal in difficulty..

In the next chapter, I will outline a revised version of PSI that attempts to maintain its strengths while correcting some of its weaknesses.

3.1.4 The Lecture Approach

PSI's appeal lies primarily in its being an alternative to the large lecture as a way of teaching substantial numbers of students. However, the lecture approach also has strengths which need to be recognized, although these get weakened quite a bit when the number of individuals being lectured to gets large. First of all, it is easily used to give explanations of complex concepts, often much much more easily than attempting to put the same explanation in print. Secondly, the teacher can monitor when students are understanding and when they are not, find out why they are not, and present material in a different way. Also, if the teacher interacts with students, he or she can stimulate a great deal of development. The teacher can ask questions and immediately challenge answers, introducing conflict and apparent contradictions in a con-

structive way. Students can be required to articulate their understandings and can receive immediate feedback. Similar explanations do not have to be repeated many times. A question asked by one student is often shared by many. Hence it can be answered for all of them at once. Analogies to previous learning can be tailored "on the fly" to the needs of a class as those needs arise.

However, the lecture method has weaknesses, the principal one of which is that lectures are often not interactive. When this happens, questions are not stirred up, apparent contradictions are not introduced, there is no correction of errors, and students do not have to articulate what they have learned. All of these things inhibit learning. Also, teachers vary considerably in their ability to use lectures to stimulate vital interaction. Further, in courses which have multiple sections or which have different instructors in different terms, the content and/or emphases will vary considerably from one instructor to another. Thus students come to subsequent courses with different backgrounds. In the next chapter, I will explore several possible enhancements that could be made to PSI to maintain its strengths and correct its weaknesses. This will involve incorporating some of the strengths of the lecture approach into PSI.

3.2 Computer Assisted Instruction

One method commonly advocated for improving the quality and/or effectiveness of teaching is CAI. In this section I want to take a brief look at the suitability of it for teaching Computer Science at the college or university level. Although this will not have a major impact on my final conclusions, so much time and money has been invested in CAI that a study of the possibilities for improving productivity in Computer Science would be incomplete without it.

3.3.1. Pros

Approaching the concept of CAI in a very general way, Alfred Bork discusses several "modes in which computers could be used and advantages of the computer as a learning device": (BOR)

-Interactive Learning: via "conversations with the computer.

-Individualization: the "dialogue" with the computer can be allowed to branch depending on student interest and capability.

-Experience: Students can do things with the computer rather than passively watch demonstrations or listen to lectures.

-Intellectual tool: This occurs primarily through programming. E.g., in teaching Physics, programming Newton's Laws of Motion then running the resulting program with various data is an effective way to learn these laws.

-Student control of Pacing: The idea here is essentially the same as PSI.

-Time and sequence control: The student can interrupt a flow of material and can jump ahead or back as needed.

-Student control over content: Students are allowed some freedom of selection of topics within constraints established by the instructor.

-Testing as a learning mode: This is the same as Bowles' use of testing in his PSI course. Tests become an aid to attaining mastery, not only an evaluation.

-Management: The computer maintains all the course records.

-Communication: The principal idea here is one of using electronic mail as a means of communication between students and the instructor.

-Personal factors: For various reasons some students may prefer an impersonal mode of learning.

It is worthwhile to note that the objectives of each of these modes is very similar to the objectives of PSI. Thus it seems reasonable to think that they could work well together.

A number of studies of CAI effectiveness have been done. One summary of these (KUL) looked at 59 different studies. These examined four modes of computer use: tutoring, course management (testing, record keeping), simulation, and students solving problems by programming. 54 of these compared examination performance of students in CBI (computer-based instruction) and conventional instruction. In 37 of the 54 studies, CBI students performed better; in 17, conventionally taught students performed better. However, only 14 of these studies showed statistically significant differences; of these 13 favored CBI. Instruction time, however, showed a clearer benefit due to CBI. Eight investigators examined time spent in instruction in the two methods. All eight showed a substantial savings in time for CBI: "On the average the conventional approach required three and one half hours per week of instructional time and the computer based approach required about two and one quarter hours." (KUL ,p.6) This is especially interesting to Computer Science educators today since there is an excess of students relative to available instructor time. Another summary (LN, p.6) reports that "Two major findings arise from the collected

research to date: computer- assisted instruction results in improved achievement when used as a supplement to traditional instruction; drill and practice is the most consistently effective mode when compared to traditional classroom instruction." The authors add, "The utility of both these statements is diluted by the knowledge that there is not much research reported on the achievement obtained by traditional classroom instruction (whatever that is!) and that drill-and-practice CAI is also the most researched treatment."

Kozma, Belle, and Williams (KBW) point out that "CAI is particularly effective in presenting content which requires practice for mastery or comprehension." They also note that computer graphics are a particularly effective way to use computers to assist in instruction.

3.2.2 Cons

In spite of all these positive results, there are a number of problems associated with CAI. For one thing, it tends to be very expensive. In the past, hardware was a major expense. With the advent of microcomputers, that cost has declined considerably. Even so, a fully stand-alone microcomputer has a number of disadvantages for CAI. There is no electronic mail capability, no help calls to a lab assistant through the machine, no automatic quiz generation and no automatic grading and record keeping; i.e.,

computer managed instruction is not possible. These problems can be overcome by configuring microcomputers so that they share a single large disk. However, this increases the expense. Moreover, programmer time is very expensive. For example, Donald Bitzer, originator of the PLATO system, estimates that one hour of courseware requires an experienced author 50 hours to prepare (SUG). Also, for reasons not fully understood, there has been considerable market resistance to CAI. Except for drill and practice, there does not exist substantial evidence that it is superior to traditional instruction. Furthermore, it seems to work well for learners who are independent, high in curiosity, and low in anxiety; it does not work as well for others. (KBW)

3.3 Conclusions

As mentioned earlier, Kulick (KUL) cites four forms of CAI commonly used - testing and record keeping, simulation, tutorials, and problem-solving via programming. All of these can be successfully used in Computer Science education. However, for the purposes of this thesis, simulation is the least relevant since the courses which have the largest enrollments are typically the introductory programming courses. In these, the experience of writing programs not just the use of prewritten simulations is most needed (although the writing of simulations them-

selves could be a useful programming exercise). Simulation could be of considerable value in more advanced courses, however; for example, in the teaching of finite state machines.

The other three aspects of CAI could certainly be successfully used in Computer Science education. Bowles' PSI course has already implemented testing and record keeping and problem-solving via programming. (In fact any programming course uses the latter.) The evidence cited above as to the effectiveness of CAI confirms that these are useful additions to Bowles' course, and would be to any PSI course. Thus the one additional thing that CAI could add to PSI is the use of tutorials.

Tutorials are currently widely used in Computer Science; for example, the tutorials on UNIX in the use of the editor and the file system. These typically use drill and practice and are effective although they tend to be boring. They could be incorporated into any PSI course; use of a tutorial could be assigned as easily as reading is assigned. The principal problem is the lack of good tutorial software in programming languages. This is an area where further work needs to be done.

CHAPTER 4

CONCLUSIONS AND RECOMMENDATIONS

My principal conclusions and recommendations are that a modified PSI format be used to teach certain courses with large enrollments and that diagnostic tests be given before students enter their first programming course. Also I recommend that the teaching guidelines that are suggested by Piaget's and Chomsky's work and which are listed in Chapter 2 be used in teaching Computer Science. I contend these steps will increase the productivity and effectiveness of Computer Science education by a modest factor. This chapter details this and other conclusions. The appendices include materials on a modified PSI course which is designed in such a way that it attempts to be consistent with the teaching guidelines suggested by Piaget's and Chomsky's work.

Section 1 summarizes the organization of a modified PSI course. This combines the separate sections of Chapter 3 which describe PSI as originally planned by Keller, as implemented by Bowles, and which detail the enhancements recommended here. Section 2 deals with the applicability of such an organization and the benefits which can be expected from it. Section 3 spells out the second major recommendation of this thesis -- that

diagnostic testing be used to identify students who need remedial work in formal reasoning before entering programming courses. Section 4 consists of final conclusions and some recommendations for future directions.

4.1 The Organization of Modified PSI Courses

Following are the main features of the modified PSI course I am recommending. For the rationale for these recommendations, see Sections 3.1 through 3.3.

1. All students need to be given written materials which are thorough and cover everything they need to know to successfully complete the course. This would normally include published materials plus supplements detailing course organization and any feature of the local computer installation not spelled out in the published materials. The teacher needs to be especially careful that homework assignments are not only drill and practice but provide challenges to student's thinking since classroom interaction to stimulate thinking is less available. Testing also needs to emphasize thinking as students will emphasize in their study the kind of things they are tested on.

2. Students should be permitted to move at their own pace, taking quizzes and submitting labs when they are ready except that they may not do either later than dead-

lines which are established before the beginning of the course and available to all students. The deadlines serve to establish a minimum pace and exist to correct the problem of the high rate of attrition and incompletion that often occurs in PSI courses. I taught a modified PSI course during the summer of 1981 with ten students and distributed a course schedule which included closing dates for submission of homework and labs and the taking of quizzes. I served as the proctor. Generally about half the students turned in their homework or took their quiz on the last possible date and half did so earlier. Homework or quizzes not completed by that date received a zero, labs a reduced score for a week, after which they received a zero. No student missed a quiz or homework due date. One student who was not doing well dropped out shortly before the end of the term. She attributed her failure to being a non-reader and to previous bad experiences with PSI which had given her an expectation of failure. One student missed a lab deadline due to "personal problems" but he successfully completed the course. Of the nine who finished, 7 received A's, 1 a B, and 1 a C for an average of 3.67/4.Ø. This compares with an average of 3.24 when I taught the same course in the winter quarter using the same text and the same final exam. Ten, of course, is a very small sample. But the results of the use of these deadlines was very successful with this group

and was very encouraging.

3. Lab assistants (proctors) need to be provided who will be located in or near the place where students run their programs. Their responsibilities would be to answer questions, give quizzes, and go over quizzes with students after they have completed them. They need to be available during regularly scheduled hours so that students can know when and where to find them. As a rough estimate of proctor time needed, I estimate 20-30 minutes per student per week. (Note: this estimate has no objective basis whatsoever; it is totally based on intuition. It is included only to help someone planning a course like this to have some starting point in estimating the amount of proctor time needed.)

4. Students should be required to demonstrate mastery of one unit before they proceed to the next one. This judgment is made by the proctor. "Mastery" is obviously very hard to define; one approach is that the instructor simply sets a standard that he or she is comfortable with, e.g., 85 or 90 percent on the unit test. Final grades are based on averaging the unit tests and the final exam. Units not completed by the deadline are given a zero. Such a grading system will probably mean a high proportion of A's and B's. This should be seen as a positive outcome; assuming the tests really tested what the instructor

wanted students to know, it indicates a high degree of learning.

5. The faculty member's main responsibilities are preparation and selection of course materials, supervision of proctors, handling questions the proctors cannot answer, and conducting lecture-discussion sessions.

6. The supervision and training of proctors requires considerable care on the part of the faculty member. He or she needs to meet with the proctors regularly to clarify questions about the course material and to discuss any problems that may arise. Also any attitudes or values the instructor wants to convey (such as the importance of good structure in programming) need to be clarified and talked through.

7. Although there is evidence that formal lectures do not contribute to the learning that takes place in a PSI course (EDU.,p.22), I recommend that regularly scheduled class meetings still be held. For small classes these could be used as lecture-discussions or as opportunities for small-group or individual tutoring sessions. For larger classes, they would have to be lecture-discussions. They should definitely not be lectures in the formal sense, but rather interactive sessions in which the instructor answers questions, and if necessary, "primes the pump" by presenting some aspect of the course material

to stimulate questions and thought. The purpose of these lecture-discussions is to help establish a pace for students, to provide for role-modeling and socialization, to provide some uniformity for students who are working with many different proctors, and to provide an opportunity for the instructor to communicate directly whatever values or emphases he or she feels need to be communicated. These also provide a source of input for students who do not learn well by reading. Bowles, in his Educom seminar, includes a paper by Steve Franklin, a colleague of Alfred Bork, a leading CAI proponent, and a user of PSI. Speaking of his PSI course he says, "Although the text was chosen with particular attention to its suitability for self-study, there are two hours of lecture each week on the material. Beyond the information these lectures convey, they give the class a cohesion and help students pace themselves." One or two meetings a week should suffice, but some experimentation needs to be done.

8. Students should be allowed to retake quizzes until they pass them. A reasonable waiting period before retaking should be set. (Bowles uses 18 hours.)

9. Putting testing and record keeping on the computer as Bowles has done would be a tremendous time saving for proctors and would release them to help students more. Furthermore it would enable more frequent and more

thorough testing than would otherwise be possible. Care needs to be taken, however, to protect against cheating. One way to discourage cheating is to have a traditional final exam and have it count a large proportion of the grade in the course. Careful security to prevent cheating on the final would be essential.

10. Even though an instructor lectures less often in a PSI course than in a lecture course, he or she should not be given a corresponding increase in classroom teaching load. The added responsibilities for materials preparation and for proctor training and supervision are sufficient to make for the reduced lecture responsibilities. The gain to a department by using PSI is in the capacity of a single instructor (with proctors) to handle a large number of students in a single section.

11. Consideration should be given to making video tape presentations of the course content available as an aid to students who have difficulties grasping material by reading it. This would be another optional resource available for people who want to use it.

12. As Bowles has done, lab assignments could be tested by a program written by the instructor. This program could feed the student's program sufficient test data to thoroughly test it. Proctors would then only need to check programs for style. This could yield more thorough

testing of programs and a greater emphasis on good style than is possible with entirely hand graded labs.

4.2 Applicability and Benefits

The modified PSI format described here could be successfully used with any course. However, it would provide the most obvious benefit in courses with large numbers of students as a significant reduction in the number of faculty required to staff these courses could be achieved. The principal limiting factor in its applicability will probably be the availability of competent proctors. In smaller classes, a lecture-discussion format with an instructor who interacts with students is educationally very effective. However, several features of a modified PSI course such as course assistants and/or the retaking of quizzes could be added as money, faculty time, and facilities allow.

One particular type of course is particularly noteworthy as a candidate for PSI and that is programming language courses subsequent to the first one. The background with which students enter these courses often varies widely. Furthermore, many students are able to learn subsequent languages very quickly. Thus the self-paced feature is especially attractive for these courses.

As for benefits, let us consider a concrete example. Suppose a school offers a programming course with four sections averaging 25 students per section for a total of 100 students. A single instructor allocated 50 hours per week of proctor time could handle this course in a modified PSI format and each student would get more individual help than he or she would get in a traditional lecture course. Let us estimate the faculty salary (including benefits) to cover four sections as between $12,000 and $16,000. Using PSI, the faculty salary is one-fourth that, $3000-$4000. Allowing $3.50 per hour for 15 weeks for proctors, their salary would amount to $2625. Thus the PSI approach would cut the staffing costs of this course by over 50%. More importantly in the present crisis, however, 3 extra courses of faculty time are freed up to be used elsewhere.

If more personal contact between students and instructor is really felt to be essential, the instructor could receive load credit equivalent to 2 lecture courses for this one programming course, and meet with the students weekly in three or four separate sections.

In this case, then, there is a clear gain both in faculty time and financially. The price for this is a replacement of faculty teaching by "peer" teaching. The success of PSI in general plus the evidence in the educa-

tional literature suggest that this is a valid exchange and that educational quality need not be lost in doing this.

Consider another example. At the Rochester Institute of Technology, the undergraduate course schedule for 1981-82 listed a total of 147 course offerings, counting multiple sections separately. These are made up of 122 sections for majors and 25 service course sections. Of these 47 of the sections for majors are in programming courses and 14 of the service sections are programming courses. Converting some of the service courses and non-introductory programming courses to PSI could reduce the number of sections of programming courses from 61 to 46. Assuming one faculty member with assistants could handle a PSI course and assuming a faculty full-time equivalent is 9 courses, this is a savings of 1.67 faculty FTE. Considering the original total of 147 sections, this is a savings of just over 10%. This is a relatively modest savings but is significant in light of the faculty shortage in Computer Science and the fact that Computer Science faculty nationwide only increased 2.8% (Section 1.4) between 1975 and 1979.

Financially, the savings is also significant. Suppose there are a total of 1200 enrollments in PSI courses during one academic year. Assuming 20-30 minutes per

proctor per week and an eleven week term, this means 4400 to 6600 proctor hours per year. At $3.50 per hour, this amounts to $13,200 to $19,800, substantially less than the salary required to pay 1.67 faculty FTE's.

Furthermore, the experience of being a proctor is educationally beneficial for students, is a form of financial aid, and enables students to work in their academic area rather than in non-academic on- or off-campus jobs.

Also, it must be remembered that Computer Science productivity involves not only quantity of students per FTE but quality as well. Students who complete PSI courses have consistently been shown to do better than students who complete lecture courses. Hopefully the enhancements to PSI advanced here will reduce the dropouts and incompletes typical of PSI while maintaining superior learning.

4.3 Diagnostic Testing

As discussed in Chapter 2, there is strong evidence that maturity in formal reasoning is a prerequisite to computer programming. Although some students might acquire this while learning to program, it would be better if they could develop their formal reasoning capacities first so that they could concentrate on learning programming. Thus I recommend that all students be tested for maturity in

formal reasoning before admission to their first programming course. Students who are identified as not being ready for programming should be encouraged to complete a course specifically designed to develop reasoning and problem solving skills first. Such courses are being developed in many places in the United States today and material available on them is widely available. (E.g. (COL)) Such a course would not normally be given by the Computer Science department, but would probably be taught by (or coordinated by) the Psychology Department.

A sample diagnostic test is included in the Appendix.

4.4 Conclusions and Future Directions

A principal conclusion of this thesis is that modified PSI courses and diagnostic testing can improve the productivity and effectiveness of Computer Science education. However, a number of things still need to be done.

Administratively, materials and quizzes for such courses need to be prepared. Proctors would have to be hired and trained (probably students who have already completed the course or a similar one). A facility for use as the lab would have to be selected. Testing and record keeping software needs to be written or purchased. Arrangements would need to be made with the Psychology Dept. (or whoever) to offer the necessary preparatory

courses for students diagnosed as not ready for programming. A diagnostic test would need to be selected.

From a research perspective, there is also much to be done. The hypothesis that formal reasoning is the principal prerequisite for programming needs to be investigated more thoroughly. Programs for training proctors need to be researched and developed. The assumption made in Chapter 2 that in the course of learning a computer language a student constructs something very similar to a compiler for that language in his or her own unconscious would make an excellent topic of study for someone's doctoral dissertation. Whether the PSI enhancements suggested here really do reduce dropouts and incompletes without seriously hurting quality also needs to be studied; the results of such study need to be fed back into any such courses to improve them.

APPENDIX 1

Item 1: Conservation of Displaced Volume

Students are shown a picture of two identical containers filled to the same level with water. There are two weights of the same shape, but different densities. Shown a picture of the water level displaced by the light weight in one container, students are asked to predict the level of water (higher, lower, same) displaced by the heavy weight in the other container.

Item 2: Direct Proportion - 1

Students are told that in a particular photograph a mother is 8 cm. high and her daughter is 6 cm. high. If the picture is enlarged so that the daughter is 15 cm. high, students are asked to predict the mother's height.

Item 3: Probabilistic Reasoning - 1

Four red chips and six blue chips are placed in a container on the left, while six red chips and nine blue chips are placed in a container on the right. Students are asked to predict which container they would choose (left, right, doesn't matter) to have the best chance of

drawing a red chip on the first try.

Item 4: Inverse Proportion - 1

Suppose that you are investigating the running abilities of a horse and a dog. You find that each time the horse takes a step, the dog takes a step. You measure the stride of the horse and find that it is 12 feet long. This horse can run a particular course in 30 seconds. If the dog has a four foot stride, how long will it take the dog to complete the same course?

Item 5: Inverse Proportion - 2

Suppose that you are comparing a different dog with the same horse. (Again, each time the horse takes a step, the dog takes a step.) This dog has a five foot stride. How long will it take this dog to complete the same course?

Item 6: Propositional Logic - 1

Students are asked to test the truth or falsity of the following rule: If a card has a vowel on one side, then it has an even number on the other side. Students are shown successive pictures of cards displaying E, 4, K, 7 and, in each case, asked, "Would you need to know what is on the other side of this card? -- Why?"

Item 7: Correlational Reasoning - 1

Shown a picture with 6 birds having long beaks and short tails, 2 birds having long beaks and long tails, and 6 birds having short beaks and long tails, students are asked if they think there is a relationship between the length of beak and the length of tail.

Item 8: Deductive Logic

Shown a picture of four islands, named bean, bird, fish, and snail, students are given the following clues: Clue 1: There is a way to fly between bean island and bird island. Clue 2: There is no way to fly between bird island and snail island. Clue 3: There is a way to fly between bean island and fish island. The students are asked: Is there a way to fly between bird island and fish island? (Yes, no, not enough information. Why?) Is there a way to fly between fish island and snail island? (Yes, no, not enough information. Why?)

Item 9: Separation of Variables

Students are shown four pictures: (1) a healthy plant that received a tall glass of water and light plant food, (2) an unhealthy plant that received a tall glass of water, dark plant food, and leaf lotion, (3) a healthy plant that received a small glass of water, light plant

food, and leaf lotion, and (4) an unhealthy plant that received a small glass of water and dark plant food. Told that another plant is receiving a small glass of water, light plant food, and no leaf lotion, students are asked to predict how the plant is doing.

Item 10: Correlational Reasoning - 2

Shown a picture of 8 squares with lines, 4 squares with dots, 2 circles with lines, and 1 circle with dots, students are asked if there is any relationship between the shape and the design.

Item 11: Permutations

Students are given a hypothetical situation in which 4 stores (a barber shop, a discount store, a grocery store, and a coffee shop) are to be arranged side-by-side on the ground floor of a shopping center. The students are asked to list all possible ways that they could be arranged.

Item 12: Probabilistic Reasoning - 2

Three blue chips and seven red chips are placed in a container on the left, while two blue chips and four red chips are placed in a container on the right. Students are asked which container they would choose (left, right, doesn't matter) to have the best chance of drawing a blue

chip on the first try.

Item 13: Direct Proportion - 2

Students are told that it takes 6 cups of flour (along with other ingredients) to make 4 dozen cookies and asked to predict how many cups of flour it would take to make six dozen cookies.

Item 14: Combinations

Students are told that biologists are dissecting crab stomachs to find if they are eating red, yellow, blue, or green algae (all locally plentiful) or other food. They are to list all possible varieties of algae the crabs might be eating (assuming order is not important).

Item 15: Propositional Reasoning - 2

Students are asked to test the truth or falsity of the following hypothesis: if a rat has lipids in its blood, then it will be fat. Students are asked: (i) Given blood samples with lipids, would you need to know if they came from fat or thin rats? (ii) Given blood samples with no lipids, would you need to know if they came from fat or thin rats? (iii) Given several fat rats, would you need to know if there are lipids in these rats' blood? (iv) Given several thin rats, would you need to know if there

are lipids in these rats' blood?

APPENDIX 2

The subsequent pages are the course materials for Mathematics 372, Computer Structures, offered at Nazareth College of Rochester, NY in the Spring semester of 1982. Some sample quizzes and sample lab assignments are included at the end.

The course design originated with course CS3 - Introduction to Computer Systems - of the ACM's Curriculum 78. However, because the school's program is not large enough to offer both CS3 and CS4 - Introduction to Computer Organization - Math 372 includes elements of each of them. The CS3 material on file I/O, assembler construction, and interpretive routines was omitted entirely (15% of that course). The material on macros and program segmentation and linkage (30% of the course) was shortened substantially. The material from CS4 on number representation, arithmetic, and coding (15% of that course) was added in and so was some of the material on computer architecture.

The text for the course is Programming the 6502 by Rodney Zaks. This is a very good text for this course for a number of reasons. First of all, the course is taught using Apple microcomputers and the 6502 is the Apple's microprocessor. Secondly, the book covers just the

selection of topics from CS3 and CS4 we wanted to cover. Thirdly, and perhaps most importantly, Zaks has designed his book for self-study and has written it in a style that is well-suited to the goals of this course. He scatters exercises throughout the body of the text, not just collecting them at the end of a section as is traditional. This way a student can read a portion of text which in some cases is as small as a paragraph and immediately test his or her knowledge by doing an exercise. Some of the exercises are drill but many are thought questions capable of stimulating the kind of cognitive development discussed in Chapter 2 under the implications of Piaget's theory. It is part of the design of this course that a student is required to complete all these exercises and then to have them graded by a proctor who will go over difficulties with him or her. Using this kind of an approach, I believe it is possible to teach abstract concepts using PSI and my experience in the course so far has confirmed this belief.

A number of supplementary materials have been prepared; these are included subsequently. The first is an explanation of the course organization and a course schedule. As discussed in Chapter 3, one of the principal modifications I have made to PSI is the establishment of deadlines by which units must be completed. The materials for each of the six units then follow. Each consists of a

set of behavioral objectives followed by the assignments for that unit and a collection of additional exercises. For some units these are optional, for some not. These are of three types: simple drill for situations in which I felt that additional practice with the concepts Zaks presented might be needed by some students; thought problems designed to stimulate cognitive growth ala Piaget; and syntax exercises. The latter are designed to take into account the insights gained from Chomsky's work as presented in Chapter 2. Essentially their purpose is to give students concrete experiences with each of the instructions of the language in a non-problem solving setting. My conviction here is that direct interaction with the computer while it actually executes an instruction (and variations on that instruction) is a very good way to master syntax; the effect should be same as attempting to speak with a native speaker of a language and being told immediately whether you were understood and what was understood by what you said. Next is a summary of the system utilities available for using the 6502 assembler language on the Apple.

The materials conclude with several sample quizzes and two sample labs. The labs are extremely important in that they provide an "integrative" function in the course. I.e., it is in writing programs that the concepts that were developed, the individual items of syntax that were

learned, and the problem solving skills of the individual all have to be put together.

BIBLIOGRAPHY

(BAI) G. L. Bailes,T. M. Countermine, "Computer Science (1978) Enrollment, Faculty and Recruiting", SIGCSE Bulletin, vol. 11, no. 2, p.43-51

Results of a 1978 survey of all colleges with majors in Computer Science.

(BOR) Alfred Bork, "Interactive Learning: Millikan Lecture, American Association of Physics Teachers, London, Ontario, June, 1978", Am. J. Phys. 47(1), Jan. 1979, p.5

An overview of CAI by a leading proponent.

(BOW) Kenneth L. Bowles, "Pascal and Microcomputers Seminar", Educom, 1979

The course materials prepared by Bowles for a seminar given by him on UCSD Pascal.

(CHA) Kenneth Chastain, The Development of Modern Language Skills: Theory to Practice , Center for Curriculum Development Inc., Philadelphia, Pa., 1971

A basic text for the preparation of foreign language teachers.

(CHO) Noam Chomsky, Review of B. F. Skinner's Verbal Behavior , Language, vol. 35, no. 1 (1959), p.26-58

Chomsky's refutation of the audio-lingual approach to teaching language.

(CHR1) "As Students Flock to Computer Science Courses, Colleges Scramble to Find Professors", The Chronicle of Higher Education, Feb. 9, 1981

A summary of the current situation in Computer Science and projections for roughly the next ten years.

(CHR2) "Average Faculty Salaries in 1980-81 and Percentage

Increases over 1979-80", Chronicle of Higher Education, June 15, 1981, p.5

A report on faculty salary data for 1980-81.

(COL) Francis P. Collea and Susan Nummedal, "Development of Reasoning in Science: A Course in Abstract Thinking", Dept. of Science Education, Calif. State Univ., Fullerton, Calif.

Outlines the nature of and results obtained by a course designed to improve formal reasoning.

(COM1) "Survey Finds Demand for DP'ers up 15.9%", Computerworld, July 20, 1981, p.1,6

Reports results of a mid-1981 survey of employment demand for Computer Science professionals.

(DEN) Peter J. Denning, "Eating our Seed Corn", Communications of the ACM, June 1981, vol. 24, no. 6

Denning's analysis of the prospects for the Computer Science profession as many teachers leave education for higher paying jobs in industry.

(FEL) J. Feldman et al., "Rejuvenating Experimental Computer Science - A Report to the National Science Foundation and Others", Comm. ACM 22,9 (Sept. 1979), p.497-502

Feldman's suggestion that about 25 "centers of excellence" in Computer Science be established and funded to provide state-of-the-art computers.

(FLA) J. H. Flavell, The Developmental Psychology of Jean Piaget , New York: D. Van Nostrand, 1963

The standard text on Piaget's views. Referred to by almost every writer on Piaget; in fact it is read quite a bit more than Piaget's original writings as he is extremely difficult to read.

(GMK) Ron Good, Ed Mellon, Bob Kromhout, "A Model for Facilitating Formal Thought in the College Science Student", Abstract CHED 48, 173rd ACS National Meeting, New Orleans, March 1977

Presents a test for intellectual development and presents and analyzes 24 dimensions the authors recommend be included

in courses which stimulate growth in formal reasoning.

(HAL) Suzanne B. Haley, Ronald G. Good, "Concrete and Formal Operational Thought: Implications for Introductory College Biology", The American Biology Teacher, October 1976, p.407

Two educators who have written widely on the implications of Piaget's views for Science education summarize a number of studies of the frequencies with which college students use formal reasoning.

(KBW) Kozma, Belle, and Williams, Instructional Techniques in Higher Education, Educational Technology Publications, Englewood Cliffs, N.J., 1978

A comprehensive study of instructional techniques in education and their effectiveness.

(KEL) F.S. Keller, "Good-bye, teacher...", Journal of Applied Behavior Analysis, 1968, vol. 1, p.78-89

Keller's original paper in which he presented the concept of PSI.

(KUL) James A. Kulik, Chen-Len Kulik, Peter A. Cohen, "Effectiveness of Computer-based College Teaching: A Meta-analysis of Findings", Review of Educational Research, vol. 50, no. 4, Winter 1980, p.525-44

A broad summary of the research into CAI effectiveness and analysis of the results of this research.

(KUR) Barry Kurtz, "Investigating the Relationship Between the Development of Abstract Reasoning and Performance in an Introductory Programming Class", ACM SIGCSE Bulletin, vol. 12, no. 1, Feb. 1980, p.110

Analyzes the relationship between scores on a standardized test of formal reasoning and grades in a programming course.

(LN) "CAI Effectiveness Debate Continues", Learning Notes / Winter Quarter 1970, p.6

A brief summary of research into CAI effectiveness.

(MIT) W. Mitchell, "Computer Education in the 1980's: A Somber View", SIGCSE Bulletin, vol. 12, no. 1 (Feb 1980),

The author expresses several concerns about Computer Science education and its potential growth including several he feels are not being adequately addressed.

(SUG) Robert Sugarman, "A Second Chance for Computer-aided Instruction", IEEE Spectrum, August 1978, p.29

A survey of some of the major CAI systems currently in use.

(TAU) O.E. Taulbee, S.D. Conte, "Production and Employment of Ph.D.'s in Computer Science - 1977 and 1978", Comm. ACM 22,2 (Feb. 1979), p.75-76

Data and discussion on the topics mentioned in the title.

Math 372
Computer Structures

Spring 1982

1. COURSE ORGANIZATION

This course will be taught using a modified version of the Personalized System of Instruction (PSI). What this means is that the course will be partially self paced. The course will meet twice a week on Tuesdays and Thursdays at 1:30 as scheduled. The Tuesday class will be a more or less traditional lecture-discussion class. It is strongly urged that you read the text material before coming to this class. The Thursday class will be a workshop. Questions of general interest will be discussed as a class. Any remaining time will be used to work with individuals or small groups who have specific questions. If you run into problems between classes, see me in my office, Smyth 327A, during office hours.

There will be 6 quizzes in the course, one for each unit. You may take these quizzes during the Thursday class or during any of my office hours whenever you feel you are ready. However, there is a closing date for each quiz (see below). If you want to take a quiz before the closing date, you must score an 85% before you can go on to the next unit. After the closing date you must proceed to the next unit irrespective of your score. You may retake a quiz as many times as you want up until its closing date.

There are also closing dates for homework; it may be submitted any time until that date, after which it receives a 0 grade. There is no resubmission of homework to improve its grade.

There will also be three labs. They too may be submitted any time before their closing date, after which they will also receive a zero.

A comprehensive final exam will be held during final exam week as scheduled by the registrar's office.

Weighting of these factors will be:

Final exam:	40%
Quizzes:	20%
Labs:	20%
Homework:	20%

2. ASSIGNMENTS

Units 1 and 2 of this course are conceptual - there is no programming involved. Unit 1 is concerned with how data is represented internally in a computer. These problems will be of two types - thought questions (to help you understand

the concepts involved) and drill (to give you practice with the number representations.) Unit 2 deals with the architecture of a particular microprocessor. All the problems in this unit are thought problems. Units 3-5 deal with programming a microprocessor in assembler language. Here again thought problems will be used but another type of question will also appear - the syntax exercise. These are short questions designed to enable you to master the syntax of the assembler language. They involve little writing of practical or applicable programs; however, they are extremely important if you are to learn the language. For each of them a program must be written (or an existing program modified). This program must be run on the computer and whatever debugging you need to do must be done! It is crucial that you understand why each of these programs performs as it does at each step.

The homework for a unit must be submitted before the quiz on that unit may be taken. Note that homework closing dates are a week earlier than quiz closing dates.

3. LABS

Labs must be well-organized and well-documented. Each lab should be submitted with a carefully written algorithm expressed in either pseudo-code or a flow chart. Individual sheets should be bursted and stapled with a single staple in the upper left hand corner and with the algorithm on top. If you use the thermal printer cut your "scroll" into pages so that it consists of roughly 8-½ by 11 sheets and staple them as above.

4. TEXT

The textbook is PROGRAMMING THE 6502 by Rodnay Zaks.

MTH 372
Computer Structures

Closing Dates

Unit	Homework	Test
1	1-28	2-4
2	2-11	2-18
3	3-4	3-11
4	4-1	4-8
5	4-15	4-22
6	4-29	5-4

Lab	
1	3-11
2	4-15
3	5-13

UNIT I

Math 372 Computer Structures

OBJECTIVES

By the end of this unit you should be able to:

1. Change decimal numbers to binary, octal, and hex - and the reverse;

2. give the BCD integer, BCD floating point,and ASCII representations of numbers where the representation may be more than one byte in length;

3. decode BCD and ASCII representations;

4. find the one's and two's complements of numbers and demonstrate for each that the complement of the complement is the original number;

5. perform two's complement arithmetic for integers of various numbers of bytes;

6. explain what the carry and overflow bits are and compute them for different sums;

HOMEWORK

1. Read chapter 1 in Zaks, doing each of the exercises as you come to them.

2. Do as many of the following additional practice problems as you feel are necessary for you.

EXERCISES

1. Convert the following numbers to binary:

   0
   1
   2
   4
   7
   8
   16
   25
   32
   33
   55
   127
   255

(Exercises con't.)

2. convert the following unsigned binary numbers to decimal:

   00000011

   01000001

   01010101

   10000000

3. Do the following sums in binary. Then convert to decimal to verify your results:

```
  0001010        00010110        11000000
+ 0001001      + 00101111      + 01011101
```

4. Change each of the following to binary then find its one's complement:

   0
   1
   3
   17
   91
   127

5. Find the two's complement of each of the numbers presented in the previous problem.

6. For the following subtractions, represent each number in two's complement form then do the computation. Convert your result back to decimal to verify it. Ignore the carry bit in converting back to decimal.

```
  7      0      127      39      71
 -5     -6     - 63     -43     -59
```

7. Compute the carry and the overflow bits for each of the following. Check each result for correctness.

```
  9      37      13       9      -3      10110101
 -6     +29     +20     -15     -7      10100110
```

(EXERCISES Con't.)

8. Referring to exercise 1.12 in the book (and problem 7 above if necessary), how can you tell from the C and V bits whether a result is in error?

9. Code the following as packed BCD integers using one byte:

   0
   13
   72
   99

10. What is the smallest possible one byte BCD integer? The largest? How many one byte BCD integers are possible? How many different integers can be represented using one byte in binary? Why is there such a difference?

11. Code the following using multibyte BCD:

    -17
    156
    412
    -9099

12. Write the following in binary:

    2.0
    0.5
    0.75
    10.75
    33.25
    0.3

    } for simplicity, allow 6 significant bits

13. Rewrite each of the numbers in problem 12 in normalized form.

14. Write each of the numbers in problem 12 using the floating point representation on page 30 of your text.

15. Compute the parity bit for each of the following using even parity:

    0000001
    0101010
    1110000
    1111111

16. Repeat problem 15 using odd parity.

(EXERCISES Con't.)

17. Convert the following binary numbers to octal and to hexadecimal:

    00000000
    00101001
    10110111
    00011111
    11110000

18. Convert the following hexadecimal numbers to binary and to decimal:

    1A
    27
    F3
    10
    AB

19. Convert the following octal numbers to binary and decimal:

    27
    10
    35
    16
    12

20. Perform the following hexadecimal arithmetic computations:

17	2A	6AF	37A	6A6	777	666	765
+23	98	-391	-1B7	+3AB	+4C1	-35B	-479

UNIT 2
Math 372 Computer Structures

OBJECTIVES

At the end of this chapter you should be able to:

1. Define the terms listed in question 1 below;

2. explain the function of each hardware item listed in question 2 below;

3. reproduce the block diagram of a typical microprocessor unit (page 39);

4. given the diagram of the 6502 on page 42, explain the flow of data;

5. trace an instruction through the fetch, decoding, and control, and execution cycles explaining what is happening at each stage.

METHOD

The purpose of this chapter is to familiarize you with the internal organization of the 6502, the chip which is the "brains" of the Apple microcomputer. This is not an end in itself; the 6502 is simply an example of some of the principles of organization that are used in all computers. There is a lot of terminology to master and a number of abstract concepts as well. I suggest the following steps in attacking this chapter. Try to finish steps 1-5 before the Tuesday class on the chapter.

1. read through the chapter paying special attention to any word that is italicized. Try to figure out from the context what these words mean. Mark any you don't understand so you can ask about them in class.

2. After completing the chapter, go through the list of terms below and attempt to define as many as possible (in pencil).

3. Compare your answers to the explanations given in the book; improve your explanations accordingly.

4. Read over the discussions of each term you couldn't fill in at all, then close the book and try again.

5. Repeat this process until you can produce reasonable definitions for all the terms without looking at the book.

6. Answer the thought questions for the chapter.

QUESTIONS

1. Define each of the following terms:

   boot strap (monitor) program

   status flag

   nanosecond

   fetch cycle

   decoding

   execution

   LIFO

   push

   pop

   software stack

   page

2. Explain the function of each of the following:

   MPU

   CPU

   ALU

   CU

   data bus

   address bus

   control bus

   clock

   crystal

   ROM

   RAM

   register

   interface chip (PIO)

   accumulator

   PC

   PCH

   PCL

   IR

(QUESTIONS Con't.)

.index register

stack register

hardware stack

3. Consider the instruction: ADC $2000

which instructs the 6502 to add the contents of location 2000 to the accumulator. Trace this instruction through the fetch, decoding and execution cycles explaining in detail what is happening at each stage.

4. Explain the difference between ROM and RAM.

5. How are registers different from memory?

6. To add the quantity B to the quantity A in the 6502 requires 3 instructions:

```
LDA   A   loads A to accumulator
ADC   B   adds B to accumulator
STA   A   stores result in A
```

Other computers without an accumulator, however, can do this in one instruction like this:

```
ADD  A,B.
```

Discuss the pros and cons of each of these approaches.

7. Try to infer at least one use for the stack- be as specific as possible in explaining how it would be used.

8. Discuss the pros and cons of hardware versus software stacks.

UNIT 3

Math 372 Computer Structures

OBJECTIVES

By the end of this unit you should be able to:

1. Use the editor and assembler to prepare and run programs;

2. use the monitor to directly enter data in machine code to specific memory locations or to examine memory;

3. explain the following terms: opcode, operand, comment field, non-destructive read, pseudo-operation, precision, logical operation, subroutine, parameter passing;

4. write addition, subtraction, multiplication or division programs in 6502 assembler using the algorithms given in the text for numbers one or more bytes in length and for both the BCD and decimal representations;

5. explain what each of these algorithms is doing at each step and why;

6. use the BRK instruction as an aid in debugging;

7. analyze the time an assembler routine will take to run;

8. code subroutine calls and returns including saving and passing parameters via memory, registers or the stack;

ASSIGNMENTS

1. Read chapter 3 in the book only as far as exercise 1 and write the program it requests.

2. Using the handout on the editor and the program format section of the assembler handout prepare the following program for running on the microcomputer:

```
        ORG     $2000           ;sets starting location
ADR1    EQU     $3000           ;initialize addresses
ADR2    EQU     $3001                   "
ADR3    EQU     $3002                   "
        CLC                     ;prepare for binary add
        CLD
        LDA     ADR1            ;move 1st number to AC
        ADC     ADR2            ;add second
        STA     ADR3            ;store in third
        BRK                     ;stop
```

(ASSIGNMENTS Con't.)

During the preparation of this, experiment with the editor enough that you are familiar with all its commands including SAVE and LOAD

3. Now read the sections of the assembler handout dealing with the use of the ASM command and the EQU and ORG pseudo-ops. Assemble this program.

4. Using the handout on the loader, load this program. Go on to the monitor and enter some data into locations 3000 and 3001. Check to see if it is correct. Now run the program. Note that when it halts the contents of all the registers are displayed. Display locations 3000, 3001 and 3002 to verify that the program worked correctly.

5. Now that you have a successfully running program, you can use it as the basis for trying some features of the assembler and monitor. Try the following:

   a.) Change the data in 3000 and 3001 and run it again. Verify the result. Try it with overflows and carries and explain the resulting differences in the P register.

   b.) Try out each of the different monitor functions. (With the exception of the subroutine call under I-O). Display the output on the printer then bring it back to the screen.

   c.) Go back to the editor and modify the source code so as to try out each of the readability pseudo-ops. Reassemble and see how they work.

   d.) Use the DFB pseudo-op to define ADR1, ADR2, and ADR3 instead of EQU. Note that you can initialize the contents of these locations this way.

   e.) Try initializing some sections of memory using the other data definition pseudo-ops. Examine your machine code listing to see if they worked the way you expected.

   f.) Try out the conditional assembly function by including 2 slightly different versions of the code and a flag which will enable the assembler to select one version. Then reverse the flag and assemble again.

6. Do the rest of the exercises in the text and run any programs they request. Pay careful attention to doing exercise 12 carefully.

Unit 4

OBJECTIVES

At the end of this unit you should be able to:

1. Explain what each of the 6502 instructions (except CLI, RTI, and SEI) does;
2. use each of these instructions in programming;
3. read the detailed description of the operation of each instruction (chapter 4, part 2) understanding each aspect of it except addressing;
4. assemble and disassemble programs by hand.

EXERCISES

1. Read through chapter 4, section 1 and do the exercises.
2. Write a program to test the operation of all the data move instructions. The data move instructions are: LDA,LDX, LDY, STA, STX, STY, TAX, TAY, TSX, TXA, TXS, TYA, PHA, PHP, PLA, PLP. Your program should simply list each of these instructions with an appropriate argument, separating each instruction by a BRK so that you can examine its effect. For example, your program could start like this:

```
ORG     $2000
LDA     $3000
BRK
LDX     $3000
BRK
LDY     $3000
```

After assembling this program but before running it you would then toggle some data into location 3000 .

Each time the computer stops, examine the accumulator, register, stack or relevant memory location to see what the instruction did. Repeat all or parts of this program as much as you need until you feel comfortable with the instructions.

(Exercises continued)

3. Consider the following program:

```
ORG     $2000
CLC
CLD
LDA     $3000
ADC     $3001
BRK
SEC
CLD
LDA     $3000
SBC     $3001
BRK
LDX     $3000
LDY     $3000
INX
INY
BRK
DEX
DEY
BRK
INC     $3000
BRK
DEC     $3000
BRK
```

Assemble this program and toggle same data into locations 3000 and 3001. Run the program starting at location 2000. When it stops record the data and the result (be sure to record the carry flag as well). Try it with various data until you feel comfortable with the operation of the instructions. Then toggle the machine code for the SEC instruction into location 2000. Run it again using the same data and compare your results. Toggle the code for the CLC instruction back in and also change the machine code for the CLD instruction to the code for the SED instruction. Now run your program again from location 2000.

After you have experimented with this long enough that you feel comfortable with it, go on to the portion after the BRK (starting at 200A). Go through the same process as you did with the add instruction until you feel comfortable with the subtract. Then go on the rest of the program and try out the increment and decrement instructions.

(Exercises continued)

4. Now assemble the following program:

```
        ORG     $2000
        LDA     $3100
        ORA     $3101
        BRK
        LDA     $3100
        AND     $3101
        BRK
        LDA     $3100
        EOR     $3101
        BRK
        LDA     $3100
        ASL     $3100
        BRK
        LDA     $3100
        LSR     $3100
        BRK
        LDA     $3100
        ROL     $3100
        BRK
        LDA     $3100
        ROR     $3100
        BRK
```

By toggling various data items into locations 3100 and 3101, test the operation of these instructions until you feel comfortable with them.

5. This problem is concerned with the branch instructions (except BIT). These instructions are: BCC, BCS, BEQ, BMI, BNE, BPL, BVS, BVC. Their operation is quite similar and I will not give you a program to test all of them this time. Instead consider the following program:

```
        ORG     $2000
        LDA     $3100
        ADC     $3101
        BPL     EQUAL
        BRK
EQUAL   BRK
```

(Exercises continued)

Run this program with various numbers in locations $3100 and $3101. Note that the program halts in different locations depending on the result of the sum. Try toggling in the machine code for some of the other branch instructions in place of the machine code for the BPL (10), then running it.

Thought question: When this program is assembled, what value replaces the symbol EQUAL? Why?

6. At this point, the operation of assembler instructions should be fairly clear to you. However, the following programs illustrate the remaining instructions in the 6502's repertoire with the exception of the interrupt handling instructions, which we will not be using. Read over these programs. Run them if you feel doing so would help you. If you run them save an assembly listing for use in problem 8.

a.)

```
        ORG     $2000
        LDA     $3100
        BIT     $3101
        BEQ     EQUAL
        BRK
EQUAL   BRK
```

b.)

```
        ORG     $2000
        LDA     $3100
        ADC     $3101
        NOP
        BVS     OVER
        BRK
OVER    BRK
```

Try this program first with data that gives an overflow. Then change the machine code for the NOP to the machine code for the CLV instruction. Try it again with the same data.

c.)

```
        ORG     $2000
        LDA     $3100
        CMP     $3101
        BEQ     SAME
        BRK
SAME    BRK
```

Try this with various values in $3100 and $3101. Then modify the LDA to LDX and the CMP to CPX and try it again. Try it a third time using the Y register.

(Exercises continued)

6. d.)

```
            ORG    $2000
            LDA    $3100
            BMI    MINUS
            JMP    OK
MINUS       EOR    #$FF
            ADC    #1
OK          BRK
```

Thought question: What is the purpose of this program? Try this out with ØØ and FØ in location 3100. Explain the resulting values of the AC.

e.)

```
            ORG    $2000
            JSR      SUB
            BRK
SUB         LDA    $3100
            BRK
            RTS
```

In what location does this program halt? Change the second break to a NOP. Now where does it halt? When it halts in the subroutine exactly what is on the stack? Why?

7. Write out an explanation of what is meant by the function, format, data, paths, and flags sections of the manual for the ORA (p. 161) and the RTS (p. 172) instructions.

8. Assemble the programs in exercise 6 by hand. Then assemble them on the Apple to check yourself. Run them if you feel doing so would help you.

(Exercises continued)

9. Disassemble the follwoing programs:

a.)

```
A2
Ø8
A9
3A
8D
ØØ
1Ø
2E
ØØ
1Ø
CD
Ø1
1Ø
CA
8D
Ø2
1Ø
6Ø
ØØ
```

b.)

```
F8
18
A9
17
69
12
2Ø
DA
FD
ØØ
```

Unit 5

OBJECTIVES

1. Write instructions using correct syntax for each of the 6502 addressing modes: implicit, immediate, absolute, direct (0-page), relative, indexed, pre-indexed, post-indexed and indirect;

2. use each of these in programming;

3. explain when each mode would be used.

EXERCISES

1. Explain why implied addressing is given that name.

2. Why do the BRK, PHA, PHP, PLA, PLP, RTI, and RTS instructions require 3 clock cycles?

3. For each instruction which can use immediate addressing (p. 195) write a sample instruction. Include a comment field to explain what each one does:

4. Repeat #3 for those that can use absolute addressing.

5. Repeat #4 for zero-page addressing.

6. When would it be desireable to use zero-page addressing? When would it not be desireable?

7. Repeat #3 for relative addressing.

8. Why can relative addressing only be used with branch instructions?

9. Complete the following table by filling in all the instructions that can be used in the given mode with the given register.

	X	Y
absolute indexed		
0-page indexed		

10. Write sample instructions for two of the instructions in each cell of the table of exercise 9. Include a comment explaining what the instruction is doing.

(Exercises continued)

11. Write a program which will copy all the data in page 0 to locations starting at 1000 using indexed addressing. Once you have successfully run this program modify it to move page 1 to locations starting at 1000.

12. Compare the assembled code for both programs of exercise 11. Explain all differences.

13. Suppose 4 numbers are stored at 4 different locations, say 2000, 3000, 4000 and 5000 for example. Also suppose their locations are stored in the first 8 bytes of page 0 of the computer's memory. Write a program using indexed indirect addressing (pre-indexing) which will add these numbers and put their sum in location 8. Run this program. Compute the number of clock cycles it takes to execute.

14. Suppose 4 numbers are stored sequentially somewhere in memory (starting at location 1000, for example). Also suppose their location is stored at a location in page 0 (say location 0). Modify your program of exercise 12 to add up these numbers using indirect indexed addressing (post-indexing).

15. Explain why pre and post-indexing are called what they are.

16. Write a sample JMP instruction using indirect absolute addressing.

17. Exercise 5.1 on page 204.

18. Exercise 5.2 on page 209.

19. Exercise 5.6 on page 210.

Unit 6

OBJECTIVES

You should be able to:

1. Write programs to generate pulses and delays;
2. write programs to detect a signal by polling;
3. explain the difference between serial and parallel data transfer and between synchronous and asynchronous transfer;
4. explain what is meant by "handshaking";
5. write programs to print a data buffer to an output device such as a CRT screen or a printer using wait loops.

HOMEWORK

1. Read chapter 6 in Zaks up to the section on interrupts (p. 242). Do each problem as you come to it.

USING THE SYSTEM UTILITIES

I. Initializing a disk.

Your first step in using the 6502 assembler is initializing your disks. Follow these steps:

1. Insert the system master diskette into drive 1 and turn the CRT and Apple on. The disk drive will run for about 10 seconds during this step. At this point you have access to both the Applesoft and Integer Basics and can run Basic programs if you want.

2. Type

   ```
   RUN COPY
   APPLE DISKETTE DUPLICATION PROGRAM
   ORIGINAL SLOT:   DEFAULT = 6
   ```

   (The slot is the location within the computer where the disk's controller card has been placed.)

3. Type a carriage return and the computer will respond

   ```
   DRIVE:  DEFAULT = 1
   ```

   Continue typing returns until you get the message

   ```
   -- PRESS "RETURN" KEY TO BEGIN COPY --
   ```

4. Insert the DOS toolkit disk in drive 1, and your blank disk in drive 2. Be sure the write protect tab is on the toolkit disk.

5. Press the return key. The computer will now make your disk into a copy of the DOS Toolkit.

6. When it is done copying, you will see the message

   ```
   DO YOU WISH ANOTHER COPY?
   ```

   Respond with Y or N. You should make 2 copies of the DOS toolkit so that you have a disk available for backup.

II. Use of the editor

Remove the write protect label from your Applesoft took kit diskette and boot it.

When you get the] prompt, type

```
RUN EDASM
```

The computer will respond

```
APPLE II  EDITOR - ASSEMBLER
CURRENT ASSEMBLER ID STAMP IS:
01 - Jan - 82 #000000.
```

Enter the correct date and return. When you get the : prompt you are in the editor.

This is a "line-oriented" editor, i.e., it looks upon your text as a series of lines which it numbers sequentially starting at 1. A line is defined as anything appearing between two returns. The numbers are not inserted in the file but can be used to refer to a line. If a line is deleted all subsequent lines have their numbers reduced by one. Similarly if lines are inserted, subsequent lines have their numbers increased.

There is a "help" feature in this editor. At any time you may type "?" then return. The screen will display the syntax of all the edit commands.

Following is a brief summary of all the edit commands. For a more detailed discussion, see the Assembler/Editor manual. Anything in brackets is optional. Also the lower case portions of the command name are optional.

Editing Commands

Add (linenum) - allows addition of new lines to the edit file, which is maintained in memory until you tell the editor to save it. If the line number is omitted, new lines will be added at the end. If included, new lines will be added after the number you specify. To terminate this mode, type a ctrl-D followed by a return at the beginning of a line.

COpy linenum 1 (-linenum 2) TO linenum 3 - allows you to copy 1 or more lines to just before linenum 3. If linenum 2 is omitted, only linenum 1 is copied. linenum 2 must be greater than linenum 1; note that if linenum 3 is less than linenum 1, the line numbered linenum 1 will have a new value when this command is completed.

(Editing commands continued)

Change (L1-L2(,L3-L4,etc)) string where string is of the form:
delim string1 delim string2 (delim)

and delim is a delimiter character (for example, a/).

String 1 is regarded as the old string and string 2 as the new one. When this command is executed, the computer will ask you:

ALL OR SOME (A/S)?

If you reply A, all occurrences of string 1 in your program will be replaced by string 2 within the ranges of line numbers you have specified. If you omit the ranges, the entire program will be used. If you reply some, you will be queried as to whether you want to change each occurrence. Respond with ESC to reject the change (even though it will temporarily appear changed on the screen), space to accept it, and CTRL-C to cancel all changes and return to command mode. You may use a CTRL-A as a wildcard for any character in the old string (only).

Delete linenum 1 (-linenum 2) - allows you to delete a line or range of lines.

Edit - (linenum 1 (-linenum 2)) (string)
This is the most powerful command in the editor. Since this editor is a line-oriented editor, all the commands except this one work on lines or groups of lines. The edit command allows you work on the individual characters within a single line or group of lines. You may specify a single line number or range of line numbers to work on. You may also specify a string - only lines containing that string will be opened for editing. The string may contain CTRL-A's as wildcards.

Commands to edit are all control characters; anything else you type will be taken to mean "replace the character currently under the cursor by what I just typed."

Following are the edit commands:

Return - accept the line as it now appears on the screen

CTRL-X - get out of edit mode. Don't replace the current line.

CTRL-T - truncate the current line from the cursor to the end.

CTRL-R - restore the line to what it was before this attempt to edit it

CTRL-I - insert any subsequent characters typed before the cursor. Exit this mode with a return.

right arrow - moves cursor right one space without modifying the line

left arrow - moves cursor left without modifying

CTRL-F - finds the next occurence of the character to the right of the cursor

CTRL-F - deletes the character currently under the cursor

(Editing commands continued)

Find (linenum 1 - linenum 2) (string) - enables you to list all lines containing string between the two line numbers. The wildcard may be used.

Insert linenum - allows the insertion of as many lines as you want before the given line number. Exit from insert mode by typing CTRL-D or CTRL-Q. Subsequent line numbers are changed by insertions.

List (linenum 1 - linenum 2) - lists all lines within the specified range or the whole program if the range is omitted. CTRL-C terminates the listing. Typing a space temporarily halts the listing; subsequent spaces list one line at a time. Typing any other character restarts the listing at full speed.

Print (linenum 1 - linenum 2) - same as List except that it does not display the line number with each line.

Replace linenum 1 (-linenum 2) - this command combines delete and insert modes. The specified range of lines is deleted and you are put in insert mode before the next line after the deleted ones.

Disk Commands

These three commands enable you to load or save programs. They assume slot 6 and drive 1 unless you change this assumption. Never change the slot number; if you want to work off drive 2 you must change that. (See below for how to do that)

LOAD Filename - moves the specified file from disk to your edit buffer. If you type a filename which does not exist on your disk, a null file with that name will be entered into your directory. See below for instructions on how to delete it. When loading is successfully completed you will see the message END OF DATA.

SAVE (linenum 1 - linenum 2) (filename) - enables you to save your program or part of it on disk. If you have previously done a LOAD or SAVE with a filename you may omit that name from this command. If you are entering a long program you should periodically save it so as to protect yourself from accidental loss of your entire program.

(Disk commands continued)

APPEND (linenum) filename - is used to load another file into the edit buffer in addition to whatever is currently there. If linenum is omitted, the new file is appended at the end. If it is included, everything from that line number to the end of the current buffer is overwritten by the new additions.

Operating Commands

SLot num - allows you to change the slot number. This should always be 6 for our Apples.

DRive num - num can be set as 1 or 2. It is always initialized to 1.

CATalog - gives a directory of the disk in the currently specified drive.

FILE - tells the current filename and its statistics. Also tells the current slot and drive numbers.

LENGTH - tells the size in bytes of the current edit file and how much space is remaining.

MON - puts you in the monitor. CRTL-Y followed by a return or typing EDØG will get you back to the editor. Note that if you change the memory used by the assembler/editor while in the monitor, you may crash the computer when you return to the editor. See the editor/assembler manual (p. 52) for a memory map.

NEW - clears the edit buffer

PR#1, L54P66CTRL-I80N - this strange looking command causes subsequent assembly listings to be sent to the printer in slot 1, assuming 54 printed lines to a 66 line page and an 80 character width. The CTRL-I will not appear on the screen when you type this.

TRuncON - causes subsequent listings to ignore comments (assuming comments are preceded by a space followed by a semicolon).

TRuncOFF - turns off TRuncON.

(Operating commands continued)

Tabs (tablist) (string) - when the assembler lists your program, it assumes certain standard tabs and uses the space as the tab character. this command allows you to change these.

Where linenum - returns the HEX address of the beginning of any line in the edit file. Very useful if you wish to manipulate or examine your text file directly from the monitor.

END - terminates the EDASM program. To return to the editor without reloading from disk, type CALL 3075. However, you will probably lose your edit file in the process.

III. Use of the Assembler

This program is what actually translates the program you create into machine code. To invoke it, you type the ASM command when in the editor. The assembler will then be read from disk into memory replacing the editor program and your edit buffer. Thus you can see the importance of SAVEing your program before attempting to assemble it. This is a two-pass assembler, which means it reads your program twice. The first time through, it generates a symbol table and assigns values to all line labels. The second pass uses the symbol table to generate the actual machine instructions. The assembly listing is generated in pass 2; error messages are generated in both passes. See The Appendices of the editor/assembler manual for explanations of the error messages.

The ASM Command

To assemble your program, type

```
ASM  file 1 (,objfile (,slot(,disk)))
```

where file 1 is the name of your program as saved on disk. Objfile is the name you want to give to your object file (the machine code version of your program). If you omit this, the same name as your source file will be used but " . OBJØ " will be appended to it. Slot must be 6 and thus should be omitted on our machine. Disk is either 1 or 2; if omitted, 1 is assumed. Thus, for example,

```
ASM MYPROG
```

assembles the program MYPROG.

```
ASM MYPROG,,,2
```

also assembles MYPROG but puts the result on the disk in drive 2. Note that the commas must be included to accomplish this.

To abort an assembly in process, type CTRL-C. To single step through the listing, press the space bar once the listing starts and keep pressing it to step through. Typing any other ordinary character will restart the full speed listing.

Program Format

In preparing your program, follow the 6502 syntax rules as spelled out in your text. There are a few details peculiar to this assembler that need to be mentioned, however.

Labels - may be up to 250 characters in length. 16 is a normal limit, however. Terminate the label with a space. Only letters, digits and periods may be used. Labels must begin in column 1. "Page-zero" addresses (those with value under 256 decimal) must be defined before they are referenced by the assembler.

Opcodes - you must use one space between the label and opcode. Two will be interpreted as meaning you want to skip the opcode field.

Operands - may be an expression as well as a numeric value or a symbol. Expressions are evaluated by the assembler (note!) and not at execution time. The value of the expression then replaces it for generation of machine code. The standard precedence rules for expressions do not apply - evaluation is simply left-to-right.

Comments -separate comments from operands by a space and a semicolon. Use comments freely to document your program. Any statement having and * or ; in column 1 is treated entirely as a comment.

Expressions - evaluation is done during pass 2, after all symbols have been defined. In deciding whether to use a 1 byte (zero-page) or 2 byte address, only the first simple operand in the expression is examined. Constants may be used as the expression or included in it. There are four types:

decimal - any number will be assumed to be decimal unless you label it otherwise

hexadecimal - must be preceded by a $

octal - must be preceded by @. (I recommend you avoid octal on this machine.)

string - may be up to 240 characters in length, enclosed by single quotes.

(Program Format continued)

Reserved words - the single characters A,X,Y,P and S should not be used as labels; it is recommended you not use any one-character labels.

Arithmetic Operators - you may use +, -, *, and /. However, overflow is not checked. If it occurs, the lower 16 bits are kept and the overflowed bit(s) dropped. Also parentheses may not be used in forming expressions; no blanks are allowed either.

Pseudo-operations

These are codes included in your program that, in format, look just like 6502 instructions. However, they are simply commands to the assembler and do not generate any machine code.

Pseudo-ops which enhance readability

PAGE - causes a top-of-form character to be sent to the output device in listing the program. Also causes a blank line to appear on the screen.

LST ON
LST OFF } turns the listing ON or OFF

REP numb - causes numb asterisks to be printed in your listing on a line. This is very useful in setting apart comments or subroutines in your program.

CHR character - changes the asterisk which REP prints to any other character.

SKP numb - causes numb blank lines to be printed in your listing

SBTL String - this causes the specified string to be printed at the top of each page of your listing as a subtitle.

NOTE: Do not use line labels with the above pseudo-ops.

Assembler directives

These pseudo-ops control the assembly of your program, hence are very important. All may be preceded by labels.

ORG expression - establishes the starting location for your object code. If omitted you will get a listing with no object file (which is what you normally want in the early stages of debugging a program). If expression is absolute (e.g. a constant) a new output file is generated. Normally only one such ORG occurs and that as the first line of a program. Subsequent ORGs are usually "relocatable", e.g.,

```
ORG *+5
```

This means take the current location plus 5 as the place to put the next item of assembled code.

Label EQU expression - enables you to assign a value to a label. This pseudo-op should be used extensively to improve readability. E.g., constants can be given a meaningful name and can be easily changed. Expressions which are used more than once can also be named this way.

DSECT - used to reserve an area of memory for data, pointers, etc. Until a DEND is encountered, no further object code is generated, although all labels mentioned in the DSECT (Dummy Section) will be defined. An ORG to location 0 is implicitly included at the beginning. Other ORG's may be included to change this.

DEND - terminates a DSECT

MSB ON
MSB OFF } sets the most significant bit in any subsequent ASCII characters stored to 1 for ON, 0 for OFF. For characters to be displayed correctly, the Apple II expects an MSB of 1.

Data definition directives

These pseudo-ops are used to set up data areas within your program and sometimes to initialize these areas.

ASC string - is the standard instruction used to put a string in memory. If a label is attached, it's value will be the first location in memory used by this string. The string must be preceded by a delimiter and followed by the same delimiter. If the comment is omitted, so may the closing delimiter be omitted also. The MSB pseudo op controls whether the MSB is 0 or 1. If not specified the default will be 1.

(Data definition directives continued)

DCI - same as ASC except the MSB will be 0 for all characters except the last which will have an MSB of 1.

DFB exprl (,expr2---) - defines bytes according to the values of the expressions modulo 256.

DW expression - used to define a two byte word. The low order byte is stored first and the high 8 bits are stored second. This is consistent with the format expected by the indirect addressing modes.

DDB expression - like DW except that the bytes are in the opposite order.

DS expression - reserves a group of bytes without specifying what is stored in them. All elements of the expression must be already defined since this expression is evaluated in pass 1. The number of bytes reserved equals the value of the expression.

Conditional Assembly

This is a feature most sophisticated assemblers supply. It is more unusual for a microcomputer. Essentially it allows you to write a multi-purpose program and then by altering the values of a few symbols, generate a tailor-made program at assembly time.
Three pseudo-ops are available.

DO expression - The expression is evaluated in pass 1 (thus must not contain any forward references). If the value is non-zero, assembly continues at the next statement. If it is zero, assembly is omitted until an ELSE or FIN pseudo-op is encountered.

ELSE - may only occur between a DO and FIN. It reverses the result of the evaluation done in the DO and subsequent statements are treated according to the new value.

FIN - marks the end of the conditional assembly block.

IV. The Loader

Once you have successfully assembled your program, it will reside in a disk file. Suppose your source program is called MYPROG. OBJØ. To bring it into main memory, get out of the editor/assembler. When you get the prompt, type

BLOAD object file name.

This will bring your program into memory at the ORG you specified. To execute your program, enter the monitor and follow its instructions for running a program. (See below.)

V. The Monitor

The monitor is a very low level piece of software; in fact the lowest level on this machine. It provides a limited number of instructions which allow you to read or alter the contents of specific memory locations. In other words, you can get your hands directly on any location in memory, examine it and alter it as you see fit. It is invoked either through the MON editor command or, if you are in Applesoft (with the] prompt) by typing CALL -151 (return).

The latter transfers control to location 65385_{10} (which is -151) which is where the monitor program resides. Following are some of the monitor's capabilities. For a more complete discussion, see the Apple II Reference Manual.

Examining Memory

When you enter the monitor, you will be prompted by an * and a blinking cursor.

To display the contents of any location - simply type its address followed by a return. The address consists of 4 hexadecimal characters (16 bits). Leading zeros may be omitted. A location displayed like this is "opened"; the monitor remembers the "last opened location."

To display a range of locations - typing a period followed by a location will "dump" memory from the last opened location to the location you specify. The last location displayed now becomes the last opened location.

To display a few locations - memory is organized in eight byte "cuts". Pressing the return key will display from the next location after the last opened location to the end of its cut.

(The Monitor continued)

Changing Memory

To change a specific location - open the location as above. Then, in response to the *, type a colon followed by the new value.

To change several consecutive locations - open the first one, then type a colon followed by the successive values separated by spaces. You may continue on another line by starting it with a colon. Up to 85 successive locations may be changed like this.

To move a range of memory - see the manual, p. 44.

To compare two ranges of memory - see the manual, p. 46.

Programming Aids

To run a machine language program - enter the starting address of the program followed by the letter G and a return.

To list a program - enter the starting address followed by an L and a return. The screen will display 20 lines of your machine language program and will display the assembly code which generated it!

Examining registers - type a CTRL E followed by a return. This will display the A, X, Y, P and S registers.

Changing registers - after examining registers, you may change them. Type a colon followed by the new values separated by a space. They must be in the same order as displayed. To change A, for example, only its new value need be entered. However to change P, A, X and Y all must be entered even if you don't intend to change them.

(Programming Aids continued)

I-Ø - to display any character on the screen, put its 8-bit ASCII code (with MSB equal to 1) in the accumulator. Then execute

JSR $FDED.

This is a monitor subroutine that sends the contents of the AC to the screen. To send characters to the printer, enter the monitor, type 1 CTRL-P return

Then execute your program with the same subroutine call as above. Your output will be steered to the printer. When you are back in the monitor, type

0 CTRL-P return

to restore the screen as the output device.

MTH 372
Unit 1 Quiz
Version 1

1. Represent the following in binary using two's complement notation:

   64
   -75

2. For each of the following, compute the sum, the carry and the overflow bits and indicate whether or not the 8-bit result is correct:

01000110	11000010	10110100
01101100	11110110	11011011

3. Decode the following ASCII characters:

   1000000
   0111100
   0001101

4. Perform the following hexadecimal computations:

   8A
   +19

   63
   -1C

   7A
   -8B

MTH 372
Unit 1 Quiz
Version 2

1. Express the following in binary using two's complement notation and in hexadecimal:

   51
   -52

2. Suppose numbers are expressed in packed BCD. Write the resulting binary codes for the following:

   19
   71

3. Find the sum, the carry and the overflow bits. Also indicate whether the 8-bit result is correct.

01100101	01001000	11000111
01111110	00111010	11110110

4a. Convert the following hexadecimal numbers to binary:

   A8
   7C

4b. The following are the binary representations of ASCII characters using even parity. Find the characters.

   10010101
   00001111

MTH 372
Unit 2 Quiz
Version 1

1. Explain the following terms:

   data bus

   LIFO

   stack register

2. Consider the instruction:

   LDA $3000

   which causes the contents of location 3000 to be transferred to the accumulator.

   Explain what happens with this instruction in the fetch, decode and execute cycles.

MTH 372
Unit 2 Quiz
Version 2

1. Explain the following terms:

   software stack

   page

   address bus

2. Consider the instruction:

   STA $2000

   which causes the contents of the accumulator to be stored in location 2000.

   Explain what happens with this instruction in the fetch, decoding and execute cycles.

MTH 372
Unit 3 Quiz
Version 1

You may use your textbook for this quiz.

1. Explain the purposes of the SED and CLD instructions.

2. Consider the following crude algorithm for multiplying two integers M and N:

```
Begin
Move M to the X register
Clear AC
Do for I = 1 to M
    add N to AC
End do
Store AC in Sum
End
```

Code this algorithm into 6502 assembler , ignoring the possibility of overflow.

3. Consider the 8-bit multiply routine on page 84 of your text. Determine the number of clock cycles it will take to execute this program.

Math 372
Lab 1

You are to generalize the 8 bit multiply routine on page 84 of your text as follows:

input - one 16 bit and one 8 bit number. The 16 bit number is to be located in C and C+1. The 8 bit number is in D. Either number may be positive or negative; negative numbers are represented in two's complement notation.

output - one 24 bit number located in E, E+1, and E+2. The sign bit should be the left-most bit of E and the result should be in two's complement notation.

You should submit a listing of your program (properly documented), a copy of your algorithm, either in pseudo-code or a flow chart and a filled-in copy of the following table giving the values your program computed for each input.

C	C+1	D	E	E+1	E+2
00	00	00			
00	14	06			
80	14	06			
00	14	86			
80	14	86			
00	7F	02			
00	7F	82			
7F	D6	02			
80	10	10			