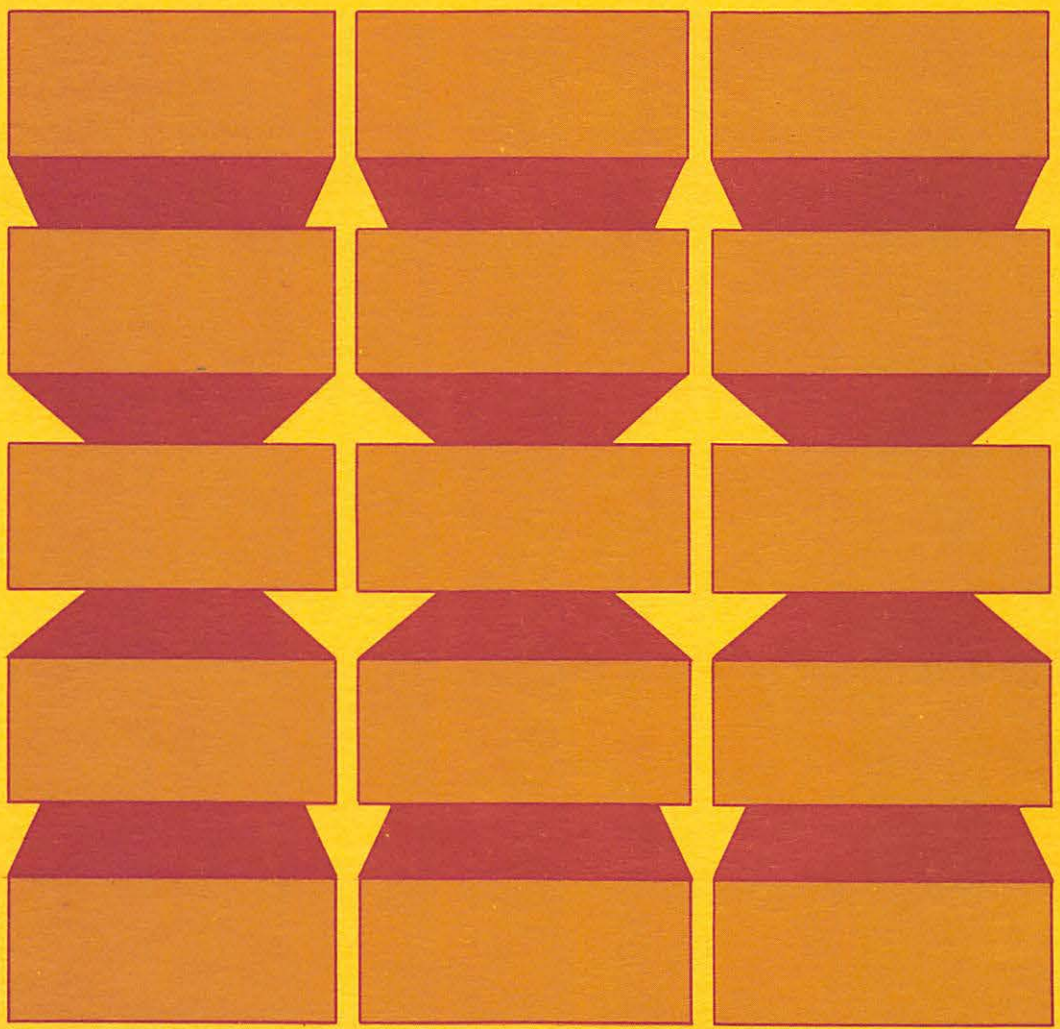
Instructor's Manual to accompany

SYSTEMS PROGRAMMING

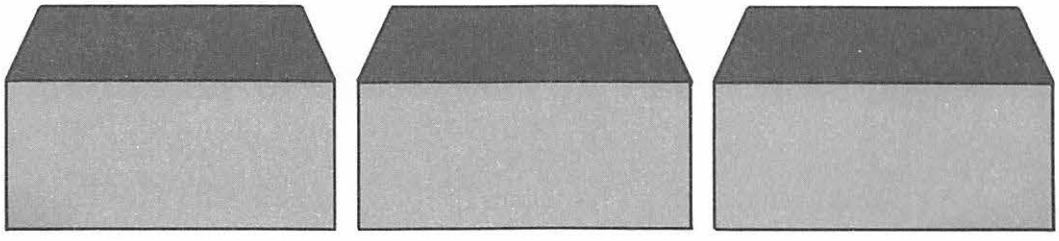
JOHN J. DONOVAN



McGRAW-HILL BOOK COMPANY

Instructor's Manual to accompany

SYSTEMS PROGRAMMING



JOHN J. DONOVAN

PROJECT M A C
MASSACHUSETTS INSTITUTE OF TECHNOLOGY
CAMBRIDGE, MASSACHUSETTS

McGRAW-HILL BOOK COMPANY
New York St. Louis San Francisco Düsseldorf London Mexico
Panama Sydney Toronto Johannesburg Kuala Lumpur
Montreal New Delhi Rio de Janeiro Singapore

Instructor's Manual to Accompany
SYSTEMS PROGRAMMING

Copyright © 1972 by McGraw-Hill, Inc. All rights reserved.
Printed in the United States of America. The contents, or
parts thereof, may be reproduced for use with
SYSTEMS PROGRAMMING,
by John Donovan
provided such reproductions bear copyright notice, but may not
be reproduced in any form for any other purpose without permission
of the publisher.

07-017604-3

3 4 5 6 7 8 9 0 W H W H 0 9 8 7 6 5 4

TABLE OF CONTENTS

I. PREFACE

We feel an excitement about the material presented in SYSTEMS PROGRAMMING; since this teacher's manual is intended to be an aid to instructors using the text, I feel I can be informal about the wording and allow the excitement to show.

We present in this manual the following aids:

(a) Possible syllabuses for presenting the material.

Through excluding or including material, this book has been used in a broad range of teaching environments. For example, the book is used in a course taught in a two-year technical school (the Lowell School), in a one-semester undergraduate course at M.I.T. and Texas Tech, in a three-semester graduate course at Northeastern and in several advanced industrial courses. However, it is primarily intended to be used as an undergraduate text, with additional material included to challange the particularly bright students if they wish to go further on their own. There is actually too much material in this book for a one-semester course, so that the instructor must be selective. The example syllabuses may be used directly, or as a basis to develop your own.

(b) Hints in reading and presenting material in the book.

Certain figures and examples in the text are important, and these are noted in this manual. Key sections of IBM reference manuals and other supplementary material are also referred to.

The textbook uses the 370 in examples, yet most material is independent of the machine. That is, the issues in the design of loaders, compilers, etc., are presented and the 370 is used as an example. We chose the 370 or 360 (these machines are compatible) because of their widespread use and because they have features that appear in most other computers. (The 360 is the "Latin" of the computer industry.) Most students wish to and should become familiar with this machine. To demonstrate the features of compilers, we chose PL/I, yet Parts 1 and 2 of the compiler chapter are equally applicable to FORTRAN, ALGOL, COBOL, or any high level language.

The examples we have chosen, however, can be easily transferred to other machines. In fact, the chapters on high level languages, formal systems, compilers, and operating systems draw few examples from any single manufacturer. We do present in this manual an example of a "conversion" of the book so as to use a UNIVAC 1108 instead of the 370.

(c) Sample exams and problems.

Included are several one-hour quizzes that may be used in addition to questions in the book, also a sample description of a machine problem, and a listing of a grading program. The problem was designed to be run on a 360 and includes descriptions of technical items, such as job control language, as well as administrative items, such as run schedules.

(d) Solutions.

The questions in the text have evolved over the six years I have taught this course. We are pleased with the problems and feel that they not only test the students' knowledge but also raise important issues. We have written solutions to the problems that appear in the text. These solutions are meant to be helpful to the instructor in obtaining a viewpoint. We have expended a great deal of effort in preparing questions and solutions, many of which cover issues not dealt with in the text, e.g., problem 8.7 covers parsing of arithmetic statements. We want to stress, however, that a small number of the questions may have more than one solution, depending on the approach to this material. Our solutions may express a different view than yours -- occasionally ours may be wrong!

There is a tradition and excitement associated with this course as it is taught at M.I.T. First, it is one of the most highly subscribed courses at M.I.T., having an enrollment of as many as 300 students per semester. Second, I am told it is one of the most challenging courses (I receive comments such as "6.251 is a 40-hour per week DO loop). Third, it is the course on which students spend the most amount of time. Yet students unanimously agree that it is worth all their efforts, They love it. It is this student enthusiasm, the spirit of the teaching assistants,

and most especially, the relevance of the material that make this an exciting course.

Other than a course that presents the basic material in SYSTEMS PROGRAMMING, I know of no other formal way of exposing a student to the important issues of systems programming.

Please feel free to call or write me if I can help further.

II. POSSIBLE SYLLABUS -- AND USES

We feel that the book has three major uses: (1) as an undergraduate text in a one- or two-semester course on systems programming; (2) as a book for professionals; and (3) as a reference for graduate students.

More specifically, the book has been used to meet the needs of the following types of courses:

1. A first course in the undergraduate computer science curriculum (following an introductory programming course, e.g., FORTRAN, PL/I).
2. A general university service course for non-computer scientists.
3. An advanced course in software.
4. A software engineering course emphasizing practical issues.
5. An extensive review or introductory course for graduate students in computer science.

The textbook is designed primarily to be used in courses meeting any one of (or a combination of) the above needs.

As the first course in computer science (following an introductory FORTRAN or PL/I course), the course serves to give a practical "feel" for the issues in systems programming. It also serves as a base for much of the theoretical work to follow. For example, a good "feel" for recursion is given in sections on implementation of macro calls within macro definitions, and later in compiers.

The issues brought forth in theoretical courses about program schemata and parallel processing stem from our discussions of multiprocessing, I/O programming, and interrupt handling. The needs and uses of formal systems areddisucssed in Chapter 7.

As a general service course the course serves to fulfill the needs of those who want to know more than a simple FORTRAN or PL/I course can offer but do not wish to take a complete computer science sequence. It serves to inform the students of the trade-offs in different computer systems and the costs of using certain features of compilers, loaders, or assemblers.

At M.I.T. the course serves as an advanced course in software, as well as a first course. In the advanced course the student is introduced to the advanced features of compilers, e.g., implementation of pointers, block structure, etc., (Part 3 of Chapter 8). It exposes him (or her) to the present state of knowledge about operating systems, and introduces advanced concepts in formal systems, bringing the student to the point where he is prepared to do research in systems, e.g., studies is memory hierarchy, debugging systems, complexity theory, automata programming, artificial intelligence, programming languages, and compiler theory.

In a software engineering course, special emphasis can be given to the practical aspects. The student can be exposed to the 370, PL/I, and programming techniques. Our discussion of assemblers, loaders, and compilers is not abstract or "theoretical"; it is a real compiler we design.

In a review course for graduate students in computer science, especially those with limited undergraduate background, nearly all of the topics may be covered. Upon completion, the student will be competent at a graduate level in the fields of systems programming, operating systems, and programming languages.

We have found at M.I.T. that persons directing many research projects and graduate advisors encourage their new graduate students to review the material in the text or take the course (even though it is listed as an undergraduate course, we usually have about fifty graduate students). This has been one of the best ways to get a new student "up to speed".

Syllabuses

We have used all the syllabuses presented here in courses given at M.I.T., in industrial firms, and other universities. If the material is presented in a course, we recommend that in addition to the text, SYSTEMS PROGRAMMING, that the student obtain the manuals for the machines used in the course, e.g., 360 PRINCIPLES OF OPERATION, 360 ASSEMBLY LANGUAGE, PL/I REFERENCE MANUAL.

We typically grade the course by weighting the grades:

1/3 - homework
1/3 - quizzes
1/3 - final exam

A. One-Semester Service Course

The course meets the needs of students who want more than just a FORTRAN course but do not necessarily intend to become computer scientists. We chose material to cover most topics, though not perhaps in the same depth needed for an undergraduate or graduate course for computer scientists. Following is one possibility.

The problem sets refer to assigning several problems at the end of the chapters. Machine problems are problems that are to be programmed. Machine problem #1 is a 360 assembly language problem; #2 and #3 are PL/I problems.

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
Department of Electrical Engineering

One-Semester Undergraduate Service Course
(three 1-hour lectures per week)

Lecture	Topic	Readings	Prob Set	Mach Prob
1	Introduction; overview of course	Preface; Chapter 1		
2	Machine structure 6 questions	2.1	1 out	
3	Machine language	2.2-2.2.2		1 out
4	Machine language	2.2.3-2.2.4		
5	Assembly language	2.3-2.4	1 due 2 out	
6	Assembly language Assemblers	3-3.2.3		1 first run
7	Assemblers	3.2.4-3.2.5		

Lecture	Topic	Readings	Prob Set	Mach Prob
8	Searching and sorting	3.3-3.5	2 due	
9	Macros -- functions and issues	4-4.2.1		
10	Macros	4.2.2-4.2.4	3 out	
11	Loaders -- loading	5.1		1 last run
12	Loaders -- design	5.2-5.4		
13	Recursion & call. seq.	8.4	3 due	
14	Review			
15	Quiz 1			
16	High level languages	Chapter 6		2 out
17	PL/I	Chapter 6	4 out	
18	Formal systems; BNF	7-7.5		
19	Introduction to compilers	8.1		2 first run
20	Model compiler	8.1	4 due	
21	Phases of compiler; lexical and syntax	8.2-8.2.2	5 out	
22	Interpretation; optimization; storage assignment	8.2.3-8.2.5		
23	Code generation; machine dependent optimization	8.2.6-8.2.9		
24	Data structures	8.3		2 last run
25	Storage classes	8.5	5 due	
26	Block structures; Interrupts; pointers	8.7-8.11		
27	Review			
28	Quiz 2			3 out

Lecture	Topic	Readings	Prob Set	Mach Prob
29	To be announced			
30	I/O programming	9-9.1.4		3 first run
31	Interrupt structure	9.1.5-9.1.7		
32	Overview of operating systems	9.2	6 out	
33	Memory management	9.2		
34	Processor management	9.3		
35	File systems	9.5	6 due	
36	Review			3 last run

B. Undergraduate Course Intended for Computer Scientists or for Serious Computer Users.

The material serves as the first course for a computer science series (after an introductory FORTRAN, PL/I course, or programming course). As such, material is chosen to motivate the student for the theoretical work he will encounter later on, and for the serious user of computers, to emphasize the practical aspects. In particular, the syllabus differs from a survey course in the following aspects:

1) a more indepth presentation of assembly language
2) slower presentation, more lectures on 360 assembly language and PL/I
3) more lectures on examples (e.g., loaders, assemblers)
4) more emphasis on engineering aspects of software
5) elimination of highly theoretical work, e.g., grammars, parsing techniques
6) elimination of I/O programming -- operating systems (except for an overview)

The course serves well as either a one- or two-semester course. See the syllabus in next section for two semester sequence.

One-Semester Course for Computer Scientists

Lecture	Topic	Readings	Prob Set	Mach Prob
1	Introduction; overview of course	Preface; Chapter 1		
2	Machine structure; 6 questions	2.1	1 out	
3	Machine language	2.2-2.2.2		
4	Machine language	2.2.3-2.2.4		1 out
5	Assembly language	2.3-2.4	1 due	
6	Assembly language examples		2 out	
7	Assembly language; assemblers	3-3.2.3		1 first run
8	Assemblers	3.2.4-3.2.5		
9	Searching and sorting	3.3-3.6	2 due	
10	Macros -- functions and issues	4-4.2.1 4		
11	Macros	4.2.2-4.2.4	3 out	
12	Loaders -- loading schemes	5.1		1 last run
13	Loader -- examples			
14	Loaders - design	5.2-5.4		
15	Calling sequences and recursion		3 due	
16	Review			
17	Quiz 1			
18	High level languages	Chapter 6		2 out
19	PL/I	Chapter 6	4 out	
20	PL/I examples			
21	Formal systems; BNF	7-7.5		

Lecture	Topic	Readings	Prob Set	Mach Prob
22	Introduction to compilers	8.1		2 first run
23	Model compiler	8.1	4 due	
24	Phases of compiler; lexical and syntax	8.2-8.2.2	5 out	
25	Interpretation; optimization storage assignment	8.2.3-8.2.5		
26	Code generation; Mach. dependent optimization	8.2.6-8.2.9		
27	Data structures	8.3		2 last run
28	Storage classes	8.5	5 due	
29	Block structures; interrupts; pointers	8.7-8.11		
30	Review			
31	Quiz 2			3 out
32	Overview of operating systems	9.2	6 out	
33	Timesharing	9.3-9.4		
34	Review		6 due	3 last run

FIRST SEMESTER OF TWO-SEMESTER COURSE
(three 1-hour lectures per week)

Lecture	Topic	Readings	Prob Set	Mach Prob
1,2,3	Introduction; machine structure; 5 questions -- 360	Preface; Chapter 1	1 out	
4,5,6	1108 machine; machine language; assembly language	2.1		1 out
--- Special 360 or 1108 assembler session ---				
7,8,9	Machine language	2.2-2.4	1 in	
10,11, 12	360 assembler language	2.3-2.4	2 out	
13,14, 15	Design of 360 type assembler	3.1-3.2.5		1 in
16,17, 18	Searching; sorting	3.3-3.6	2 in	2 out
19,20, 21	Macros -- use	4-4.2	3 out	
22,23, 24	Macros -- implementation	4.3-4.4		
25	Quiz		3 in	
26,27, 28	Loaders	5.1	4 out	2 in
29,30, 31	Loaders -- design (calling conventions; recursion)	5.2-5.4 8.4		
32,33, 34	I/O programming; interrupts	9-9.1.4		
35,36, 37	Operating systems; memory management	9.2		
38,39, 40	Operating systems; file systems; case studies -- OS, MVT, TSS, MULTICS, etc.	9.3-9.6	4 in	
41	Final Exam			

SECOND SEMESTER OF TWO-SEMESTER COURSE
(three 1-hour lectures per week)

Lecture	Topic	Readings	Prob Set	Mach Prob
1,2,3	Importance of high level language features; BNF	Chapter 6 7-7.5	1 out	
4,5,6	PL/I	Chapter 6		1 out
7,8,9	Compiler, statement of problem, structure	8.1	1 in	
10,11, 12	Lexical phase -- reductions; syntax and interpretation	8.2-8.2.3	2 out	
13,14 15	Operation procedure	Problem 7.1, Problem 7.7, 8.2.4		1 in
16,17, 18	Storage assignment; code generation; machine dependent optimization	8.2.5-8.2.6	2 in	2 out
19,20, 21	Assembly phase; practical considerations; data structures	8.2.7-8.2.9 8.3	3 out	
22,23, 24	Storage classes; block structure	8.4, 8.7		
25,26, 27	Block structure; recursion; review	8.7		
28	Quiz		3 in	
29,30, 31	Conditions; pointer interface with operating systems	8.8-8.11		2 in
32,33, 34	Formal systems	Chapter 7	4 out	
35,36, 37	Operating systems; storage management; interrupts, segmentation	9.1-9.2		

Lecture	Topic	Readings	Prob Set	Mach Prob
38,39 40	Processor management; file systems; interpreters	9.3-9.6	4 in	
41	Final Exam			

C. Graduate Course or Industrial Course for Mature Students

Material is selected to bring the graduate student to a point where he can do independent research in the field of systems programming. The material can be given in two semesters (a pace that I like), following the previous outline. (We have, in fact, given the material in a three-semester graduate course and still wished we had more time.). The material can also be given as an intense one-semester course as follows. Note that more time is spent on operating systems than in A or B.

INTENSE ONE-SEMESTER GRADUATE COURSE

Lecture	Topic	Readings	Prob Set	Mach Prob
Optional lectures on basic background information to be given as extra sessions or as first few lectures		IBM Manuals Chapters 1, 2, and 7		
1,2,3	Overview of course; Review - 360 machine language, assembly language	Chapters 1 and 2	1 out	1 out
4,5,6	Design of assembler	Chapter 3	1 in	
7,8,9	Searching and sorting	Chapters 3 and 4	2 out	
10,11, 12	Design of macro processors	Chapter 4	2 in	1 in
13,14, 15	Loaders, calling sequence	Chapter 5	3 out	
16,17, 18	Loaders and review			

Lecture	Topic	Readings	Prob Set	Mach Prob
19	**Quiz 1**			
20,21 22	High level languages, PL/I	Chapter 6	4 out	2 out
23,24, 25	Compilers	Chapter 6		
26,27, 28	Compilers	Chapter 6		
29,30 31	I/O programming	9-9.1.4	5 out	
32,33, 34	Operating systems; memory management; paging; segmentation	9.2		
35,36 37	Processor management; multiprocessing; deadly embrace	9.3		
38,39, 40	Information management; file systems	9.5	5 in	2 in
41	Quiz 2			

III. HELPFUL HINTS IN PRESENTING THIS MATERIAL

Many of the examples of the first few chapters use the 370. The student should learn how to program the 370 or 360 in assembly language. Most M.I.T. students taking this course do not have any assembly language programming background. Therefore, they learn while taking this course. We use Chapter 2 as an introduction to assembly language and then assign a machine problem, such as problem 15 of Chapter 2.

We inform the student of the pertinent sections (for a first reading) of the IBM assembly language manual (Form C28-6514).

In this manual, as in most of IBM's manuals, there is a lot of information that is unnecessary to struggle with on the first reading. I have outlined below what we feel to be the important parts to grasp on the first time through this IBM manual.

1) Section 1 (p 3-5)
2) Section 2 (p 7-18)
3) Section 3 (p 19-21,
   p 23 up to "CSECT -- Identify Control Section",
   p 26 from "SYMBOLIC LINKAGES" through p 28)
4) Section 4 (p 29-33)
5) Section 5 (p 35-38 up to "Bit-Length Specification",
   p 40 from "Operand Subfield 4: Constant" through
   p 43 up to "Floating Point Constants -- E and D"
   p 45 from "Address Constants" through p 57)

We have found that reading Chapter 2 and doing one machine problem gives enough exposure to the 360 to allow the student to cope successfully with the remaining material in the book.

We run into a similar problem when we teach compilers in that we use the PL/I language in examples. We've chosen PL/I because it has features that are complicated, and these features will exist in high level languages of the future. Also, with IBM's backing, PL/I itself should become the language of the future.

However, Parts 1 and 2 of the compiler chapter are applicable to languages such as FORTRAN or ALGOL. Only Part 3 discusses the compilation of the advanced features of PL/I. So you may choose not to introduce PL/I and use FORTRAN or any other high level language as the base of the compiler material.

If you choose to use PL/I and your students are not familiar with it (as most of ours are not) we suggest assigning Chapter 6 and problem to be programmed in PL/I. We usually have either several in-class lectures on PL/I or else post-class seminars. Sometimes Chapter 6 and a brief discussion of the differences between FORTRAN and PL/I is all that is necessary.

Other helpful handouts are sample assembly language programs and sample PL/I programs with an explanation of their listings.

The presentation in the book depends heavily on examples. The examples that are depicted in the figures on pages 69 through 73 of the text present the entire assembly process for a sample program; that is, these figures depict the contents of each data base and their transformations.

We have presented several different loading schemes for loaders and used examples to demonstrate them. Figures 5.12, 5.13, 5.14, and 5.15 contain an excellent example of a direct linking loading scheme.

The topic of compilers is a difficult and complicated one. Perhaps as an over-reaction to many so-called "compiler" books that focus unduly on parsing techniques, which are frankly not the crucial problems of modern compilers, I have deliberately tried to expose the pertinent issues, e.g., storage assignment, code generation, etc., and have left parsing as an exercise. (See problems 1 and 7 of Chapter 7.)

The compiler chapter was divided into three parts in order to make it digestible -- one sitting would be just too much.

Parts 1 and 2 use the example "MINI-PL/I" program of Figure 8.1 to reveal the relevant issues. Almost all the figures in these two parts depict the contents of various data bases used in compiling this program. Figure 8.13 gives an overview of a compiler. This figure is important and should be kept in mind throughout the chapter. Figure 8.19 is particularly helpful, and the students should work through it to see the operation of the compiler.

Note that the MINI-PL/I is very close to PL/I and FORTRAN. In order to make the example MINI-PL/I subroutine seem meaningful, we chose to have arguments. Also for simplicity and to keep a FORTRAN program flavor, we chose all variables to be STATIC. Full PL/I, of course, would ignore the conflicting static declaration for the

variables passed as arguments, as storage for these variables would not be in the program. This is a minor point, as the purpose of Parts 1 and 2 is to present the issues of compilers. Part 3 goes into details of compilation of advanced features of PL/I. The last section of Appendix B clarifies the argument issue for PL/I.

The three parts should be read serially, but in quantums. That is, Part 1 should be thoroughly understood before continuing on to Parts 2 and 3.

You may point out to the students that the design procedure outlined in section 3.1 [i.e., 1) statement or problem; 2) definition of data bases; 3) format of data bases; 4) algorithms; 5) modularity] and followed for the design of assemblers in Chapter 3, and again for the design of loaders in Chapter 5, was also followed in the design of each phase of the compiler in Part 2.

Chapter 9 has also been divided into parts to make a very long, complicated subject digestible.

The topic of the chapter is operating systems. The concept of presenting an operating system as a resource manager is one that has made possible a coherent presentation of operating systems. Parts 2, 3, 4, and 5 present methods used in operating systems for managing the resources of memory, processors, devices, and information respectively. Our examples come from a variety of operating systems and manufacturers.

Part 1 exposes the basic issues of I/O programming, interrupt handling, and asynchronous processing, which must be understood before the details of memory, processor, device, or information management can be understood.

One must know what an interrupt is and how it is handled to appreciate the cost or complexity it poses to an operating system. As has been the procedure throughout the book, we present examples. In Part 1, Figure 9.9 and pages 364 and 365 are especially important in that they expose the interaction between a CPU and I/O channel and their respective main program, interrupt handler, and I/O program.

I find it an effective educational device to divide the class in half and have the left side be the CPU and the right side be the I/O channel. Each person starts executing instructions sequentially.

The left side starts executing the main program. I draw on the board a PSW, and CSW, and a CAW. They tell me how to modify it. When the left side hits the SIO instruction, they start the right side executing the I/O program. Then both sides operate asynchronously until the I/O channel (right side) interrupts the CPU. To simulate the interrupt, I have the I/O channel student get up and poke the appropriate CPU student.

Conversion of Chapter 2 to 1108:

The following is an example of how Chapter 2 and (in a similar manner) the rest of the book can be converted to other machines. I have taught this course using a UNIVAC 1108, PDP-10, IBM 7094, IBM 1130, IBM 360/370, FOX1, Honeywell 200, GE 645.

Following are the answers to questions of section 2.1 of the UNIVAC 1108.

(1) Memory

basic unit = 36 bits (2 parity); size = 65k-256k

other units = (format 6, 9, 12, 18, 36, 72 bits)

(2) Registers

15 index

4 common

16 accumulators

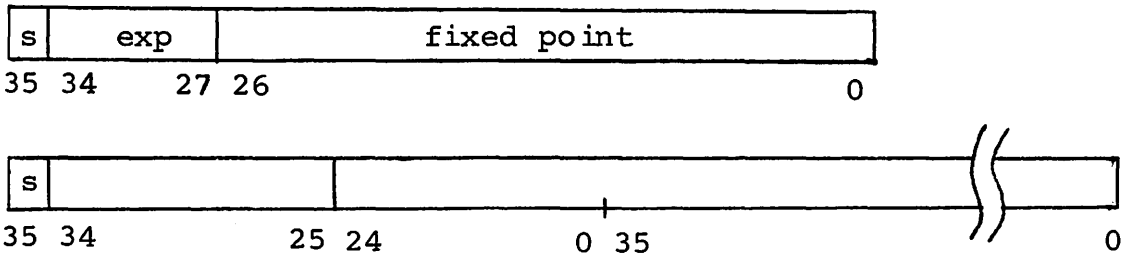
16 R-regs special (e.g., clock, counter, etc.)

(3) Data Format

Fixed: 1's complement

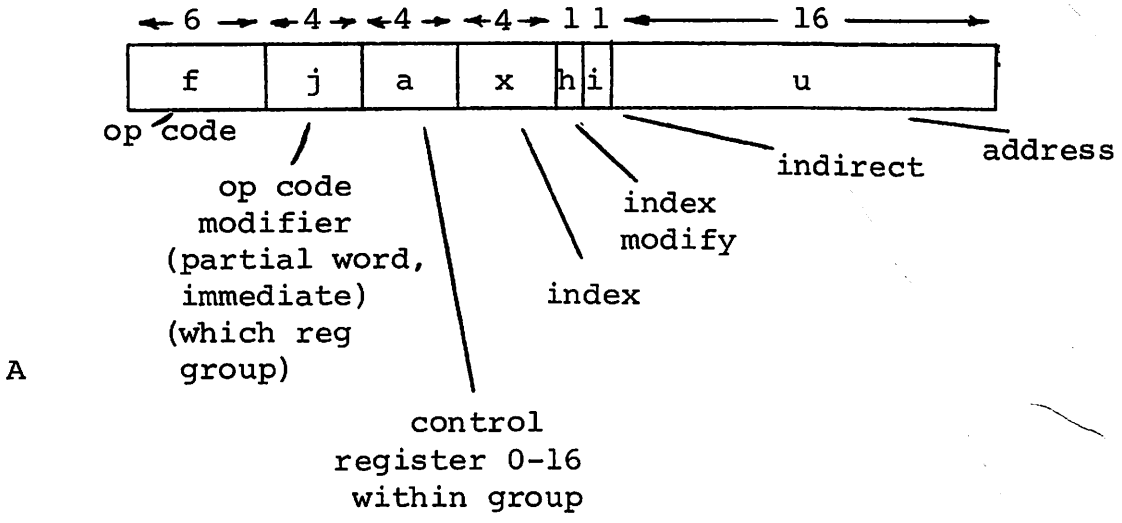
[36 bit, 72 bit, and others]

Floating:

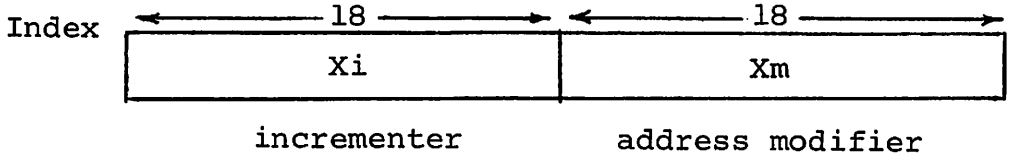


Partial Fixed: (half, etc.)

(4) Instruction



A



Effective Addressing : (u + c(X)m)* indirected

[if H = 1, Xm ← Xm + Xi
after performing op code]

CONVERSION OF EXAMPLE PROGRAM TO ADD 49 TO 10 LOCATIONS IN MEMORY

1108 CODE

Long Way

Location	Instruction		Assumption
100	LA	12,420	420 → data
101	AA	12,400	400 → 49
102	SA	12,420	
103	LA	12,421	
104	AA	12,400	
105	SA	12,421	
	⋮		

TREATING INSTRUCTIONS AS DATA

Location	Instruction		Assumption
100	LA	12,420	420 → data
101	AA	12,420	400 → 49
102	SA	12,420	401 → 1
103	LA	12,100	402 → 10
104	AA	12,401	A017 12,1
105	SA	12,100	
106	LA	12,102	
107	AA	12,401	
108	SA	12,102	
109	LA	12,402	
110	ANA	12,401	
111	JNZ	100	

immediate

Short Way

Location	Instruction		Assumption
100	LX	11,402	420 → data
101	LA	12,420,11	400 → 49
102	AA	12,400	401 → 1
103	SA	12,420,11	402 → 9
104	JGD	11,101	

IV. EXAMPLE EXAMS AND EXTRA PROBLEMS

The following are possible closed book one-hour quizzes. The first covers assemblers and loaders (Chapters 1 through 5). The second covers compilers (Chapters 6 and 8).

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
Department of Electrical Engineering

6.251 Digital Computer Programming Systems 20 Oct. 72
Prof. J. J. Donovan

QUIZ 1 Name_____

T.A._____

General Procedures

a) The examination is closed book.
b) No smoking!
c) Read each problem completely before starting to answer it. Note point values assigned to each problem. The total for the exam is 100 points.
d) Your answers should be brief but complete. It is more advantageous to answer all the questions, briefly outlining the major points, than to answer one or two in great detail.
e) All answers should be written in this exam booklet.
f) Write your name on each page.

Question	Points	
I	25 _____	(assemblers and macros)
II	25 _____	(assemblers)
III	50 _____	(loaders)

Question 1 (360 assembly language and macros) (25 points)

1) Briefly describe what will be left in register five (5) after the execution of each of the following instructions:

```
a)   L     5,ALPHA
b)   LA    5,ALPHA
c)   L     5,=A(ALPHA)
d)   LA    5,=A(ALPHA)
e)   LA    5,246
```

2) A macro is defined in lines 1 through 4 below. Write down the one line of assembly language that will be generated by the call to the macro in line 5.

```
1            MACRO
2            DELTA   &X
3            L   1,&X
4            MEND
5            DELTA   LOC
6    LOC     DC      F'10'
7            END
```

Question 2 (Assemblers) (25 points)

Consider the data bases of a 360-type assembler (without a macro facility) as presented in class.

Check the appropriate boxes in the table below. Should any of your answers require qualification, footnote as necessary

Source Text							
Object Code							
Symbol Table							
Literal Table							
Op Table							
Pseudo-op Table							
Kitchen Table							
Base Register Table							
Location Counter							
	Perm-anent	Create	Alter	Refer-ence	Create	Alter	Refer-ence
		During PASS 1			During PASS 2		

Question 3 (loaders) (50 points)

Consider the following assembly language program:

```
                                    Rel. Addr.          TXT
MIT     START     0
        EXTRN     PAUL
        BALR      15,0
        USING     *,15
        L         1,=A(ALL)
        BR        14
ALL     DC        A(PAUL)
        END
```

FIGURE 1

1) You act as the assembler. Complete Figure 1 by putting in the TXT (text) output of the assembler along with the relative address associated with that text. (The TXT should be of the form used in class; i.e., L 1,ADD is translated to L 1,disp(0,base) , DC F'10' is translated to a full word containing the number 10.

2) a. What information should appear on ESD (external symbol dictionary) cards produced by an assembler for a direct linking loading scheme?

   b. Produce the ESD card(s) for the program in Figure 1. (The format is not important).

3) a. What information should appear on RLD (relocation dictionary) cards produced by an assembler for a direct linking loading scheme?

   b. Produce the RLD card(s) for the program in Figure 1. (The format is not important).

4) You are to act as the loader and fill in the blanks (_____) in Figure 2. The program in Figure 1 should start at memory location 100. PAUL has already been loaded and has a value of 390. (There are seven (7) blanks to be filled in).

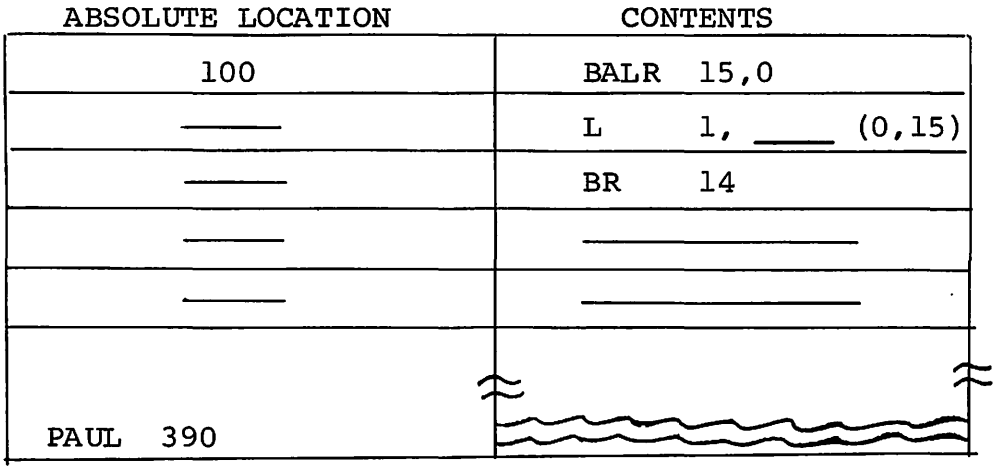


Figure 2

QUIZ 2 (same cover sheet as QUIZ 1)

Question 1 (short answer questions) (50 points)

1) Which phases of the compiler put information into or delete information from the Identifier Table? Give an example for each phase you've chosen, of what information they place into the Table.

   In addition to those above, which phases reference the Identifier Table?

2) Which phase puts the attributes for literals into the Literal Table?

   Which other phases add information to or delete information from the Literal Table? Explain.

3) Suppose we put the lexical, syntax, and interpretation phases all into one pass. How would these phases interact? Consider which phase would act as the main program and which as the subroutines.

   What is the advantage of not having the lexical analysis phase as a separate first pass?

4) Suppose we had a compiler for PL/I and wanted to change it, by modifying the data bases, so that it could compile a new language on the same machine. Which of the following data bases would have to be changed (in format or content) and why?

   a. Uniform Symbol Table
   b. Terminal Symbol Table
   c. Reductions
   d. Action Routines
   e. Matrix and Code Productions (neither or both)
   f. Identifier Table

5) What is a major disadvantage associated with optimization of the object code?

6) What criteria would you use to determine how much optimization your compiler is to perform?

7) A common type of machine dependent optimization is the elimination of unnecessary STORE's and LOAD's. Consider the following unoptimized Assembly Language code which resulted from the assignment statement C = D * (A+B);

MATRIX			ASSEMBLY LANGUAGE	
:			:	
+	A	B	L	1,A
*	D	M1	A	1,B
=	C	M2	ST	1,M1
:			L	1,M1
			M	0,D
			ST	1,M2
			L	1,M2
			ST	1,C
			:	

After some machine dependent optimization the assembly language code might look like this:

```
.
L     1,A
A     1,B
M     0,D
ST    1,C
:
```

a. On what basis did the production for the "+" operator decide not to generate the ST 1,M1 ?

b. Looking at our more efficient code we notice that we do not need temporary storage locations M1 and M2 for this assignment statement. The suggestion is made that when the machine dependent optimization part of code productions eliminates the ST 1,M1 and L 1,M1, it could also go to the Identifier Table and delete the temporary M1 (likewise for M2).

Do you agree with the suggestions?
If so, you are not very practical.
If not, give possible difficulties which could arise were this suggestion to be followed.

Question 2 (15 points)

A grammar is said to be ambiguous if it contains a sentence for which two distinct syntax trees exist.

Which of the following grammars is ambiguous? Prove this by finding a sentence with two distinct syntax trees.

1.
⟨exp⟩::= ⟨term⟩ | ⟨exp⟩+⟨term⟩ |⟨exp⟩ * ⟨term⟩
⟨term⟩::= ⟨factor⟩ | ⟨term⟩ * ⟨factor⟩
⟨factor⟩::= (⟨exp⟩) | ⟨idn⟩
⟨idn⟩::= A | B | ... | Z

2.
⟨exp⟩::= ⟨term⟩ | ⟨exp⟩+⟨term⟩
⟨term⟩::= ⟨factor⟩ | ⟨term⟩ * ⟨factor⟩ | ⟨term⟩ | ⟨factor⟩
⟨factor⟩::= (⟨exp⟩) | ⟨idn⟩
⟨idn⟩::= A | B | ... |Z

Question 3

We wish to add an IF...THEN... statement to the subset of PL/I discussed in class. It is to be of the following format:

IF (⟨idn⟩ .EQ ⟨idn⟩) THEN ⟨assignment statement⟩ ;

meaning the assignment statement is to be executed if the values of the two identifiers are equal.

1) What additions will you make to your Terminal Symbol Table?

2) You are to devise a method for implementing the IF...THEN... statement given the following restrictions:

   i. You are to add only one action routine, called "if_then", which in turn may call any of the other action routines.

   ii. The only matrix entires if_then can make are of the following format. (It need not use all of these):

OPN	OP1	OP2	
JMP	Li		Jump unconditionally to Li
JMPT	Li	Mi	Jump to Li if result of matrix line Mi is true
JMPF	Li	Mi	Jump to Li if result of matrix line Mi is false
EQUAL	OP1	OP2	Result of this matrix line has value true if the values of OP1 and OP2 are equal
DEF	Li		Define Li as "here"

where the Li are labels created by if_then.

For example, the matrix entry

```
DEF L3
```

might result in the assembly language code

```
L3  EQU  *
```

a) State in English what additions you would make to the reductions.
b) Describe exactly what your action routine if_then will do when it is called. (Hint: If PROCEDURE A calls PROCEDURE B, which then calls PROCEDURE C, then C would normally return through B).
c) Illustrate your solution by producuing the intermediate language and the assembly language for the following statement:

```
IF (A .EQ B) THEN X = W+Y ;
```

V. EXAMPLE MACHINE PROBLEM AND GRADING PROBLEM

We have included an example machine problem write-up and the listing of a corresponding grading program. It is not necessary to use a grading program, but for large classes we find a grading program is helpful to us, as well as to the students. The grading program does all the I/O, error handling, supplies the student's program with test data, and grades the student.

The following is a listing of the grading program for problem 15 of Chapter 2. It is actually a more general grading problem, because by changing the tables at the end under the TEST CASES heading, the program may be used to grade such variations of problem 15 as "convert Roman numerals given in a register to integers" "count the number of A's in some register", etc.

The grading program calls the students' program, supplies it with data, handles most errors, and gives debugging aids, e.g., core dump of his program, and grades the student.

This sample grading program is intended to be simple and easy to use. For very small classes, we have not used a grading program. For classes with more than 100 students, such as at M.I.T., we have developed a more comprehensive and efficient OS/360 Class Monitor System (CMS). This system requires considerably more effort to install at a computer facility.

The last page of the listings is a sample student program with appropriate control cards.

SYSTEMS PROGRAMMING COURSE

Professor J. J. Donovan
Stuart E. Madnick

Assigned: November 3, 1971
Due: December 1, 1971

Machine Problem 1

Assignment: Do Machine Problem 2.15 in DONOVAN.
(Count number of EBCDIC commas in registers 2-3, return count in register 1).

Notes:

1. Specific instructions describing control cards needed and other operational details will be provided later.

2. On the due date, 1 December 1971, you should submit the computer printout of your most successful run including the listing of your program and the text cases.
NOTE: partial credit will be awarded as long as your program is commented and you explain the results obtained.

3. On the following pages you will find sample printout for various interesting examples.

4. No smoking allowed while punching cards.

```
FILE: 6251     SYSIN    P1                MASSACHUSETTS INSTITUTE OF TECHNOLOGY

//GRADER JOB   (M4568,2286,5,1000,50),MSGLEVEL=1
//       EXEC  ASMC
//C.SYSGO  DD  DSNAME=SIX251,DISP=(NEW,KEEP),SPACE=(TRK,(1,1)),        X
//             UNIT=1C0,DCB=(RECFM=FB,BLKSIZE=1600,LRECL=80),          X
//             VOLUME=SER=CCSYDR
//C.SYSIN  DD  *
GRAD     TITLE 'GRADING PROGRAM - FEB. 2,1968 - WA TAYLOR, SE MADNICK'  GRA00010
         MACRO                                                          GRA00020
         TDFN  &N,&M                                                    GRA00030
         LCLA  &A                                                       GRA00040
         ORG   DTRBL+&N                                                 GRA00050
&A       SETA  &M-&N                                                    GRA00060
         DC    &A.C'.'                                                  GRA00070
         MEND                                                           GRA00080
         MACRO                                                          GRA00090
&NAME    GPRINT &ADDR,&LENGTH                                           GRA00100
&NAME    LA    3,&ADDR                                                  GRA00110
         LA    4,&LENGTH                                                GRA00120
         BAL   14,PRINT      ******   DESTROYS GR'S 1-4   ******       GRA00130
         MEND                                                           GRA00140
         EJECT                                                          GRA00150
*        PORTION OF THE PROGRAM THAT IS INVARIANT BETWEEN PROBLEMS.     GRA00160
*        THIS SECTION PRONTS OUT MESSAGES GIVING THE LOAD POINTS AND    GRA00170
*        THE NUMBER OF TRIALS THE STUDENT WILL GET.                     GRA00180
         SPACE 2                                                        GRA00190
         ENTRY DUMPME                                                   GRA00200
GRADER   START 0                                                        GRA00210
         SAVE  (14,12),,*                                               GRA00220
         BALR  BASE,0                                                   GRA00230
         USING *,BASE                                                   GRA00240
         L     BASE,AGRADER  MOVE BASE TO THIS POINT                    GRA00250
         USING GRADER,BASE                                              GRA00260
```

```
*                                                                      GRA00270
         LR     11,13                                                  GRA00280
         LA     13,SAVEAREA                                            GRA00290
         ST     11,4(13)                                               GRA00300
         ST     13,8(11)                                               GRA00310
         MVC    BUFFER,BUFFER-1          BLANK OUTBUFFER INITIALLY     GRA00320
         GETPOOL                PDCB,40,131                            GRA00330
         OPEN   (PDCB,OUTPUT)                                          GRA00340
*                                                                      GRA00350
         GPRINT MESS1,L'MESS1                                          GRA00360
*        SEE IF STUDENT IS THERE                                       GRA00370
         L      1,VSTUDENT      GET ADDRESS OF HIM                     GRA00380
         LTR    1,1             SEE IF IT IS ZERO                      GRA00390
         BNZ    GOKIT           HE IS THERE                            GRA00400
         GPRINT                 MESNOSTU,L'MESNOSTU                    GRA00410
         B      HOM             FLUSHIM                                GRA00420
GOKIT    UNPK   HEX(7),AGRADER+1(4)                                    GRA00430
         TR     HEX(6),HEXTBL                                          GRA00440
         MVC    MESS2B(6),HEX                                          GRA00450
         UNPK   HEX(7),VSTUDENT+1(4)                                   GRA00460
         TR     HEX(6),HEXTBL                                          GRA00470
         MVC    MESS3B(6),HEX                                          GRA00480
         GPRINT MESS2A,MESS3D-MESS2A                                   GRA00490
         L      1,TESTNO        GET NUMBER OF CALLS TO STUCENT         GRA00500
         STC    1,MESNO         PUT IT IN MESSAGE                      GRA00510
         OI     MESNO,X'F0'     PLACE NUMBER OF TESTS IN MESSAGE       GRA00520
         GPRINT MESS4,MESS4E-MESS4 TELL HOW MANY TRIALS                GRA00530
         B      GRAGON          BRANCH AROUND MESSAGES USED ONCE.      GRA00540
         SPACE 2                                                       GRA00550
******    OUTPUT MESSAGES     ******                                   GRA00560
         SPACE 2                                                       GRA00570
MESS1    DC     C'0 THIS IS THE 6.251 MONITOR SPEAKING.  HELLO.'       GRA00580
MESS2A   DC     C'0 I AM LOADED AT '                                   GRA00590
MESS2B   DC     C'XXXXXX'                                              GRA00600
MESS2C   DC     C' (HEX)'                                              GRA00610
MESS3A   DC     C', AND YOU ARE LOADED AT '                            GRA00620
MESS3B   DC     C'XXXXXX'                                              GRA00630
MESS3C   DC     C' (HEX).'                                             GRA00640
MESS3D   EQU    *               MARK END OF EESSAGE                    GRA00650
```

```
MESS4    DC    C'0 YOU WILL BE GIVEN '                                    GRA00660
MESNO    DC    C'X'                                                       GRA00670
         DC    C' PROBLEMS TO SOLVE.  GOOD LUCK.'                         GRA00680
MESS4E   EQU   *                                                         GRA00690
MESNOSTU DC    C'0 YOUR PROGRAM CANNOT BE ENTERED.  YOU EITHER OMITTED XGRA00700
               TO NAME THE START INSTRUCTION, OR MISNAMED IT.'           GRA00710
         EJECT                                                           GRA00720
*        HAVE NOW ISSUED MESSAGES SAYING MONITOR IS IN CONTROL, AND      GRA00730
*        TOLD HOW MANY TESTS THERE WILL BE.                              GRA00740
*        THE NEXT SECTION OF THE PROGRAM WILL BE SPECIFIC TO THE ROBLEM  GRA00750
*                                                                        GRA00760
*        THE ASSUMPTION HAS BEEN MADE THER THERE WILL BE SUFFICIENT      GRA00770
*        REGISTERS FOR ONE CALLED 'COUNT' TO BE USEDTO COUNT             GRA00780
*        THE TRIALS.  ALL SUBROUTINES THAT NEED TO KNOW THE TRIAL NUMBE  GRA00790
*        USE THIS REGISTER.                                              GRA00800
*                                                                        GRA00810
*        THE PROGRAM STUOUT IS A SUBROUTINE TO WRITE MESSAGES SAYING     GRA00820
*        THAT CCNTROL IS BEING TRANSFERRED TO THE STUDENT, INITIALIZE    GRA00830
*        TIMER AND OPERATION INTERRUPTS, AND CALL THE STUDENT.  THE      GRA00840
*        GENERAL REGISTERS ON ENTRY TO THE STUDENT WILL BE THE SAME      GRA00850
*        AS ON ENTRY TO STUOUT, EXCEPT FOR 14 AND 15.  ON RETURN, THE    GRA00860
*        REGISTERS AS THEY WERE WHEN THE STUDENT WAS CALLED WILL BE      GRA00870
*        STORED 0-15 IN 'PSAVEREG',.  THE REGISTERS AS THEY WERE WHEN    GRA00880
*        THE STUDENT RETURNED WILL BE STORED 0-15 IN 'STUREG'.  ALL      GRA00890
*        REGISTERS EXCEPT 'BASE' WILL BE AS THEY WERE WHEN THE STUDENT   GRA00900
*        RETURNED CONTROL.  REGISTER 1 WILL HAVE BEEN STORED IN          GRA00910
*        LOCATION 'STUANS'.  IF YOU DO NOT NEED ANYTHING BUT 'STUANS',   GRA00920
*        DO A LM 0,15,PSAVEREG , AND THINGS WILL BE AS THEY WERE BEFORE  GRA00930
*        THE CALL TO STUOUT.  YOU CAN THEN ANALYZE THE RESULTS.          GRA00940
*                                                                        GRA00950
*        INTERNAL SUBROUTINES AVALIABLE:                                 GRA00960
*                                                                        GRA00970
*        DUMP  DUMPS STUDENT AREA USING 'STUREG' FOR REGISTERS.          GRA00980
*              PROBLEM-  DESTROUS MOST OF THE REGISTERS                  GRA00990
*        DUMPMESS          DUMPS STUDENT USING 'STUREG' BUT WILL NOT     GRA01000
*                          HARM REGISTERS                                GRA01010
*        DUMPME            EXTERNAL ENTRY, DUMPS REGISTERS AS THEY       GRA01020
*                          WERE ON ITS CALL                              GRA01030
```

```
         EJECT                                                          GRA01040
******   SPECIFIC SECTION OF PROGRAM    ******                         GRA01050
         SPACE 2                                                        GRA01060
GRAGON   SR     COUNT,COUNT                                            GRA01070
LOOP     LA     COUNT,1(0,COUNT)                                       GRA01080
         BAL    14,NUMBER      PRINT OUT TRIAL NUMBER                  GRA01090
         LR     COUNTX,COUNT                                           GRA01100
         SLL    COUNTX,3                                               GRA01110
         LA     COUNTX,TESTS-8(COUNTX)                                 GRA01120
         MVC    MESS6B+1(8),0(COUNTX)                                  GRA01130
         GPRINT MESS6A,MESSY-MESS6A                                    GRA01140
         LM     2,3,0(COUNTX)                                          GRA01150
         BAL    14,STUOUT      GO AND TRY STUDENT                      GRA01160
         LM     0,13,PSAVEREG                                          GRA01170
         UNPK   HEX(9),STUANS(5)                                       GRA01180
         TR     HEX(8),HEXTBL                                          GRA01190
         MVC    MESS10B,HEX                                            GRA01200
         GPRINT MESS10A,MESS11-MESS10A                                 GRA01210
         LR     COUNTX,COUNT                                           GRA01220
         SLL    COUNTX,2                                               GRA01230
         LA     COUNTX,TESTSA-4(COUNTX)                                GRA01240
         CLC    STUANS,0(COUNTX)                                       GRA01250
         BNE    WRONG                                                  GRA01260
RIGHT    GPRINT MESS11,L'MESS11                                        GRA01270
         B      TRY                                                    GRA01280
WRONG    GPRINT                MESS12,L'MESS12                         GRA01290
         BAL    14,DUMPMESS    DUMP CORE ALL OVER THE PLACE            GRA01300
TRY      C    COUNT,TESTNO     SEE IF DONE ALL TESTING                 GRA01310
         BNE    LOOP                                                   GRA01320
HOM      LA     13,SAVEAREA    RECOVER ADDRESS IN CASE OF LOSS         GRA01330
         L      13,4(13)                                               GRA01340
         RETURN                (14,12),T                               GRA01350
         SPACE 2                                                       GRA01360
MESS6A   DC     C'0 CONTENTS OF REGISTERS 2 AND 3 ... '                GRA01370
MESS6B   DC     C'''XXXXXXXX'''                                        GRA01380
MESS6C   DC     C' (EBCDIC).'                                          GRA01390
MESSY    EQU    *              END OF MESSAGE                          GRA01400
MESS10A  DC     C'0 CONTENTS OF REGISTER 1 ... '                       GRA01410
MESS10B  DC     C'XXXXXXXX'                                            GRA01420
```

```
MESS10C  DC    C' (HEXADECIMAL).'                                    GRA01430
MESS11   DC    C'0 CONGRATULATIONS, YOUR ANSWER IS CORRECT.'         GRA01440
MESS12   DC    C'0 SORRY, YOUR ANSWER IS INCORRECT.'                 GRA01450
         EJECT                                                       GRA01460
******     ROTINE TO PRINT TRIAL NUMBER    ******                  GRA01470
         SPACE 2                                                     GRA01480
NUMBER   ST    14,DM14        SAVE REGISTER 14                      GRA01490
         STC   COUNT,MESS5B   GET COUNT INTO MESSAGE                GRA01500
         OI    MESS5B,X'F0'   TURN IT INTO A CHARACTER               GRA01510
         GPRINT MESS5A,MESS5E-MESS5A                                 GRA01520
         L     14,DM14                                               GRA01530
         BR    14             RETURN                                 GRA01540
         SPACE 2                                                     GRA01550
******   NUMBER MESSAGE  ******                                      GRA01560
         SPACE 2                                                     GRA01570
MESS5A   DC    C'1 THIS IS TRIAL NUMBER '                            GRA01580
MESS5B   DC    C'X'                                                  GRA01590
         DC    C'.'                                                  GRA01600
MESS5E   EQU   *              MARK END OF MESSAGE                    GRA01610
         EJECT                                                       GRA01620
******   CALLS STUDENT   ******                                      GRA01630
         SPACE 2                                                     GRA01640
STUOUT   STM   0,15,PSAVEREG SAVE ALL REGISTERS                     GRA01650
         ST    14,DM14        SAVE RETURN                           GRA01660
         SPIE  PRMINT,((1,15))     *** ISSUE SPIE ***               GRA01670
         GPRINT MESS7,L'MESS7                                        GRA01680
         GPRINT SPACE,L'SPACE                                        GRA01690
         STIMER  TASK,TIMINT,BINTVL=SECOND   *** SET TIMER ***       GRA01700
         LM    0,5,PSAVEREG  ONLY UP TO 5 IS HARMED BY PRINT         GRA01710
         L     15,VSTUDENT                                           GRA01720
         BALR  14,15                                                 GRA01730
         SPACE 2                                                     GRA01740
*        CONTROL TRANSFERRED TO STUDENT AND RETURNS HERE             GRA01750
         SPACE 2                                                     GRA01760
         USING *,14                                                  GRA01770
         STM   0,15,STUREG                                           GRA01780
         L     BASE,AGRADER                                          GRA01790
         USING GRADER,BASE                                           GRA01800
```

```
         DROP  14                                                      GRA01810
         ST    1,STUANS                                                GRA01820
         STM   0,6,SAVEREG   SAVE THAT WHICH MAY CHANGE                GRA01830
         TTIMER               CANCEL    *** TURN OFF TIMER ***       GRA01840
         SPIE  ,              *** TURN OFF SPIE ***                   GRA01850
         GPRINT MESS9,L'MESS9                                         GRA01860
         LM    0,6,SAVEREG   RESTORE THAT WHICH MAY HAVE CHANGED      GRA01870
         L     14,DM14                                                GRA01880
         BR    14            GO BACK                                  GRA01890
         SPACE 2                                                      GRA01900
******   STUTOU MESSAGES   ******                                     GRA01910
         SPACE 2                                                      GRA01920
MESS7    DC    C'0 CONTROL HAS BEEN PASSED TO YOU.'                   GRA01930
MESS9    DC    C'0 I AM BACK IN CONTROL.  YOU DID NOT BLOW UP.'       GRA01940
         EJECT                                                        GRA01950
******    PROCESS PROGRAM INTERRUPT IN STUDENT PROGRAM   ******      GRA01960
         SPACE 2                                                      GRA01970
PRMINT   EQU   *                                                      GRA01980
         DROP  BASE                                                   GRA01990
         USING *,15                                                   GRA02000
         STM   2,13,STUREG+2*4                                        GRA02010
         MVC   STUREG(8),20(1)                                        GRA02020
         MVC   STUREG+14*4(8),12(1)                                   GRA02030
         L     BASE,AGRADER                                           GRA02040
         USING GRADER,BASE                                            GRA02050
         DROP  15                                                     GRA02060
         UNPK  HEX(9),4(5,1)                                          GRA02070
         TR    HEX(8),HEXTBL                                          GRA02080
         MVC   PSW(8),HEX                                             GRA02090
         UNPK  HEX(9),8(5,1)                                          GRA02100
         TR    HEX(8),HEXTBL                                          GRA02110
         MVC   PSW+8(8),HEX                                           GRA02120
         GPRINT               PRGINTM,L'PRGINTM                       GRA02130
         GPRINT               PRGPSW,RUNSTOP-PRGPSW                   GRA02140
         GPRINT               RUNSTOP,L'RUNSTOP                       GRA02150
         LA    14,HOM         FAKE A BAL                              GRA02160
         B     DUMP                    GO HOM AFTER IT                GRA02170
         SPACE 2                                                      GRA02180
******  INTERRUPT MESSAGES  ******                                    GRA02190
```

```
         SPACE 2                                                          GRA02200
PRGINTM  DC    C'0 YOU WERE INTERRUPTED BY THE HARDWARE.  TOO BAD.'       GRA02210
PRGPSW   DC    C'0 PROGRAM INTERRUPT PSW ... '                           GRA02220
PSW      DC    C'XXXXXXXXXXXXXXXX'                                       GRA02230
PRGPSW2  DC    C' (HEX)'                                                 GRA02240
RUNSTOP  DC    C'0 CONTROL WAS GIVEN BACK TO ME.'                        GRA02250
         EJECT                                                           GRA02260
******     PROCESS TIMER INTERRUPT IN STUDENT PROGRAM     ******         GRA02270
         SPACE 3                                                         GRA02280
TIMINT   EQU   *                                                         GRA02290
         DROP  BASE                                                      GRA02300
         USING *,15                                                      GRA02310
         STM   0,15,STUREG                                               GRA02320
         L     BASE,AGRADER                                              GRA02330
         USING GRADER,BASE                                               GRA02340
         DROP  15                                                        GRA02350
         SPIE  ,                *** TURN OFF SPIE ***                    GRA02360
         GPRINT                 TIMMES1,L'TIMMES1                        GRA02370
         GPRINT                 RUNSTOP,L'RUNSTOP                        GRA02380
         GPRINT                 NOREGS,L'NOREGS                          GRA02390
         LA    14,HOM          FAKE A BAL                                GRA02400
         B     DUMP            GOHOM AFTER                               GRA02410
         SPACE 2                                                         GRA02420
******  TIMER MESSAGES  ******                                           GRA02430
         SPACE 2                                                         GRA02440
TIMMES1  DC    C'0 YOU TOOK TOO MUCH TIME, PROBABLE ENDLESS LOOP.'       GRA02450
NOREGS   DC  C'0 WARNING - REGISTERS PRINTED BELOW MAY BE INCORRECT.'    GRA02460
         EJECT                                                           GRA02470
******  EXTERNAL DUMP PROGRAM  ******                                    GRA02480
         SPACE 2                                                         GRA02490
DUMPME   BALR  15,0            SET THE REGISTER IN CASE STUDENT DID NOT  GRA02500
         USING *,15                                                      GRA02510
         DROP  BASE                                                      GRA02520
         STM   0,15,STUREG     SAVVE WHERE WILL BE DUMPED                GRA02530
         L     BASE,AGRADER                                              GRA02540
         USING GRADER,BASE                                               GRA02550
         DROP  15                                                        GRA02560
         GPRINT                 CLDME,L'CLDME                            GRA02570
```

```
         BAL   14,DUMP                                                  GRA02580
         LM    0,15,STUREG                                              GRA02590
         BR    14              GO BACK                                  GRA02600
         SPACE 2                                                        GRA02610
CLDME    DC    C'0 YOU ASKED FOR THIS DUMP.  I WILL RETURN TO YOU.'     GRA02620
         SPACE 3                                                        GRA02630
******   INTERNAL STUDENT DUMP   ******                                 GRA02640
         SPACE 2                                                        GRA02650
DUMPMESS STM   0,15,PSAVEREG SAVE FROM OBLIVION                         GRA02660
         BAL   14,DUMP                                                  GRA02670
         LM    0,15,PSAVEREG                                            GRA02680
         BR    14              GOHOM                                    GRA02690
         EJECT                                                          GRA02700
******    INTERNAL DUMP PROGRAM    ******                              GRA02710
         SPACE 2                                                        GRA02720
DUMP     ST    14,DM14                                                  GRA02730
         GPRINT                MESS13,L'MESS13                          GRA02740
         GPRINT                SPACE,1                                  GRA02750
         L     1,=V(STUDENTN)                                           GRA02760
         ST    1,LASTSTU                                                GRA02770
         LTR   1,1                                                      GRA02780
         BNZ   OK                                                       GRA02790
         L     1,STARTAD                                                GRA02800
         LA    1,4*8*40(1)                                              GRA02810
         ST    1,LASTSTU                                                GRA02820
         GPRINT                NOLAST,L'NOLAST                          GRA02830
         GPRINT                SPACE,1                                  GRA02840
OK       MVC   DUMPOUT(L'REG07),REG07                                   GRA02850
         MVI   EBCDUMP-1,X'5C'                                          GRA02860
         MVI   EBCDUMP+32,X'5C'                                         GRA02870
         LA    1,STUREG                                                 GRA02880
         BAL   14,DUMPR                                                 GRA02890
         MVC   DUMPOUT(L'REG815),REG815                                 GRA02900
         LA    1,STUREG+8*4                                             GRA02910
         BAL   14,DUMPR                                                 GRA02920
         GPRINT                SPACE,1                                  GRA02930
         MVC   DUMPOUT(L'DUMPHD),DUMPHD                                 GRA02940
         GPRINT                DUMPOUT,L'DUMPHD                         GRA02950
         MVC   DUMPOUT(L'REG815),DUMPOUTX        BLANK BUFFER           GRA02960
```

```
         SR    COUNT,COUNT                                                GRA02970
         B     DUMPLP+4                                                   GRA02980
DUMPLP   LA    COUNT,4*8(COUNT)                                           GRA02990
         LR    COUNTX,COUNT                                               GRA03000
         A     COUNTX,STARTAD                                             GRA03010
         ST    COUNTX,TEMP                                                GRA03020
         UNPK  HEX(7),TEMP(5)                                             GRA03030
         TR    HEX(6),HEXTBL                                              GRA03040
         MVC   ABS(6),HEX                                                 GRA03050
         LR    1,COUNTX                                                   GRA03060
         BAL   14,DUMPR                                                   GRA03070
         C     COUNTX,LASTSTU                                             GRA03080
         BL    DUMPLP                                                     GRA03090
         GPRINT NOTE,L'NOTE  TELL THEM ALL ABOUT IT                       GRA03100
         GPRINT               SPACE,1                                     GRA03110
         L     14,DM14                                                    GRA03120
         BR    14            GOHOM                                        GRA03130
         SPACE 2                                                          GRA03140
DUMPR    ST    14,DREG                                                    GRA03150
         MVC   EBCDUMP(32),0(1)        MOVE INTO CHARACTER DUMP AREA      GRA03160
         TR    EBCDUMP(32),DTRBL       REPLACE ILLEGAL HEX WITH C'.'      GRA03170
         LR    COUNTD1,1                                                  GRA03180
         LA    COUNTD2,4*8-4(COUNTD1)                                     GRA03190
         LA    COUNTP1,HEXDUMP                                            GRA03200
         B     DUMPRL+8                                                   GRA03210
DUMPRL   LA    COUNTD1,4(COUNTD1)                                         GRA03220
         LA    COUNTP1,10(COUNTP1)                                        GRA03230
         UNPK  HEX(9),0(5,COUNTD1)                                        GRA03240
         TR    HEX(8),HEXTBL                                              GRA03250
         MVC   0(8,COUNTP1),HEX                                           GRA03260
         CR    COUNTD1,COUNTD2                                            GRA03270
         BNE   DUMPRL                                                     GRA03280
         GPRINT               DUMPOUTX,L'DUMPOUTX                         GRA03290
         L     14,DREG                                                    GRA03300
         BR    14                                                         GRA03310
         SPACE 2                                                          GRA03320
******   DUMP PROGRAM MESSAGES  ******                                    GRA03330
         SPACE 2                                                          GRA03340
```

```
MESS13   DC    C'0 I AM DUMPING YOUR PORTION OF MEMORY.'                GRA03350
REG07    DC    CL14' REG. 0-7'                                         GRA03360
REG815   DC    CL14' REG. 8-15'                                        GRA03370
DUMPHD   DC    C'  ABS.'                                               GRA03380
NOLAST   DC    C'0 YOU DID NOT SUPPLY AN ENTRY NAMED ''STUDENTN''.'    GRA03390
NOTE     DC    C'0 I CHOOSE TO PRINT ILLEGAL CHARACTERS AS PERIODS.'   GRA03400
         SPACE 2                                                       GRA03410
******   TRANSLATE TABLE FOR EBCDIC DUMP   ******                      GRA03420
         SPACE 2                                                       GRA03430
DTRBL    DC    256AL1(*-DTRBL)          SET FOR NULL TRANSLATION       GRA03440
         TDFN  0,64                                                    GRA03450
         TDFN  65,74                                                   GRA03460
         TDFN  81,90                                                   GRA03470
         TDFN  98,107                                                  GRA03480
         TDFN  112,122                                                 GRA03490
         TDFN  128,193                                                 GRA03500
         TDFN  202,209                                                 GRA03510
         TDFN  218,226                                                 GRA03520
         TDFN  234,240                                                 GRA03530
         TDFN  250,256                                                 GRA03540
         EJECT                                                         GRA03550
******    INTERNAL GPRINT ROUTINE    ******                            GRA03560
         SPACE 2                                                       GRA03570
PRINT    ST    14,SV14        SAVE RETURN                              GRA03580
         BCTR  4,0                                                     GRA03590
         EX    4,MOVE                                                  GRA03600
         PUT   PDCB,BUFFER                                             GRA03610
         MVC   BUFFER,BUFFER-1                                         GRA03620
         L     14,SV14        RESTORE RETURN                           GRA03630
         BR    14                                                      GRA03640
         EJECT                                                         GRA03650
******    CONSTANTS AND VARIABLE STORAGE    ******                    GRA03660
         SPACE 2                                                       GRA03670
*        'STARTAD' IS THE ADDRESS THE DUMP PROGRAM STARTS AT.  IF IT IS GRA03680
*        NOT CHANGED BY THE PROGRAM, ONLY THE STUDENT RPROGRAM WILL BE GRA03690
*        DUMPED.  IF IT IS DESIRED TO DUP A PART OF THE GRADING PROGRAM GRA03700
*        *     AS IN THE CASE WHERE THERE ARE SUBROUTINES SUPPLIED TO  GRA03710
*        THE STUDENT, THEN IT OUGHT TO BE CHANGED TO START SOMEWHERE   GRA03720
```

```
*         ELSE.  THIS CAN BE DONE EASILY                              GRA03730
*                                                                     GRA03740
STARTAD   DC    V(STUDENT)                                            GRA03750
AGRADER   DC    A(GRADER)                                             GRA03760
VSTUDENT  DC    V(STUDENT)                                            GRA03770
HEX       DS    CL10                                                  GRA03780
MOVE      MVC   BUFFER(1),0(3)                                        GRA03790
          DC    C' '                                                  GRA03800
BUFFER    DS    CL136                                                 GRA03810
PSAVEREG  DS    16F                                                   GRA03820
SAVEAREA  DS    18F                                                   GRA03830
TESTNO    DC    F'5'            NUMBER OF TRIALS                      GRA03840
SPACE     DC    C'0'                                                  GRA03850
DM14      DS    F               SPOT WHEREIN DUMP SAVES 14            GRA03860
SV14      DS    F               SAVE REGISTER 14 UNIFORMLY            GRA03870
SECOND    DC    F'100'                                                GRA03880
SAVEREG   DS    16F                                                   GRA03890
STUREG    DS    16F                                                   GRA03900
STUANS    DS    F                                                     GRA03910
LASTSTU   DS    F                                                     GRA03920
DREG      DS    F                                                     GRA03930
TEMP      DS    F                                                     GRA03940
HEXTBLX   DC    C'0123456789ABCDEF'                                   GRA03950
HEXTBL    EQU   HEXTBLX-C'0'                                          GRA03960
DUMPOUTX  DC    CL132' '                                              GRA03970
DUMPOUT   EQU   DUMPOUTX+1                                           GRA03980
ABS       EQU   DUMPOUT                                               GRA03990
HEXDUMP   EQU   DUMPOUT+11    PUT THE HEX HERE                      GRA04000
EBCDUMP   EQU   DUMPOUT+92    BUT PUT EBEDIC HERE                   GRA04010
          EJECT                                                       GRA04020
******     TESTS CASES     ******                                     GRA04030
          SPACE 2                                                     GRA04040
TESTS     DS    0F                                                    GRA04050
          DC    CL8'AB,CDEFG'
          DC    CL8'A,BCD,EF'
          DC    CL8'ABCDEFGH'
          DC    CL8',,,,,,,,'
          DC    CL8',A,B,C,D'
```

```
         SPACE 4                                                                GRA04110
TESTSA   DS    0F                                                               GRA04120
         DC    F'1'
         DC    F'2'
         DC    F'0'
         DC    F'8'
         DC    F'4'
         DC    4F'0'          SPACE TO PREVENT DCB INTERFERENCE                 GRA04180
         EJECT                                                                  GRA04190
******    DCB    ******                                                         GRA04200
         SPACE 2                                                                GRA04210
PDCB DCB DSORG=PS,MACRF=PM,DEVD=DA,DDNAME=SYSPRINT,RECFM=FA,BLKSIZE=131         GRA04220
         EJECT                                                                  GRA04230
******    REGISTER DEFINITIONS    ******                                       GRA04240
         SPACE 2                                                                GRA04250
BASE     EQU   12                                                               GRA04260
COUNT    EQU   5                                                                GRA04270
COUNTX   EQU   6                                                                GRA04280
COUNTD1  EQU   7                                                                GRA04290
COUNTD2  EQU   8                                                                GRA04300
COUNTP1  EQU   9                                                                GRA04310
         END   GRADER                                                           GRA04320
/*
```

```
//C        EXEC   PGM=ASMBLR,PARM='NODECK,LOAD'
//SYSLIB   DD     DSNAME=SYS1.MACLIB,DISP=OLD
//SYSGO    DD     DSNAME=&TEMP,DISP=(MOD,PASS),UNIT=SYSDA,                X
//                SPACE=(3000,(50,5)),DCB=(RECFM=FB,BLKSIZE=1600,LRECL=80)
//SYSUT1   DD     DSNAME=&UT1,UNIT=SYSDA,SPACE=(3000,(50,25))
//SYSUT2   DD     DSNAME=&UT2,UNIT=SYSDA,SPACE=(3000,(50,25))
//SYSUT3   DD     DSNAME=&UT3,UNIT=SYSDA,SPACE=(3000,(50,25))
//SYSPRINT DD     SYSOUT=A
//SYSIN    DD     *
STUDENT  START  0                                                       MP100010
         ENTRY  STUDENTN                                                MP100020
         USING  *,15
         SR     1,1                                                     MP100030
         LM     4,5,=A(8,BACK)                                          MP100040
BACK     STC    3,Y
         SRDL   2,8                                                     MP100060
         O      2,Y                                                     MP100070
         EX     3,CLI                                                   MP100080
         BNE    BCTR                                                    MP100090
         LA     1,1(,1)                                                 MP100100
BCTR     BCTR   4,5                                                     MP100110
         BR     14                                                      MP100120
Y        DC     F'0'                                                    MP100130
CLI      CLI    Y,0                                                     MP100140
STUDENTN EQU    *                                                       MP100150
         END                                                            MP100160
/*
//L        EXEC   PGM=IEWL,PARM='NOLIST,NOMAP,NCAL',COND=(4,LT)
//SYSPRINT DD     SYSOUT=A
//SYSLIN   DD     DSNAME=SIX251,UNIT=1C0,DISP=OLD,VOLUME=SER=CCSYDR
//         DD     DSNAME=&TEMP,DISP=(OLD,DELETE)
//SYSLMOD  DD     DSNAME=&LMOD(USERPROG),DISP=(NEW,PASS),                 X
//                UNIT=SYSDA,SPACE=(3000,(100,50,1))
//SYSUT1   DD     DSNAME=&UT1,UNIT=SYSDA,SPACE=(3000,(50,25))
/*
//G        EXEC   PGM=*.L.SYSLMOD,COND=(4,LT)
//SYSLMOD  DD     DSNAME=&LMOD,DISP=(OLD,DELETE)
//SYSPRINT DD     SYSOUT=A
/*
```

VII. SOLUTIONS

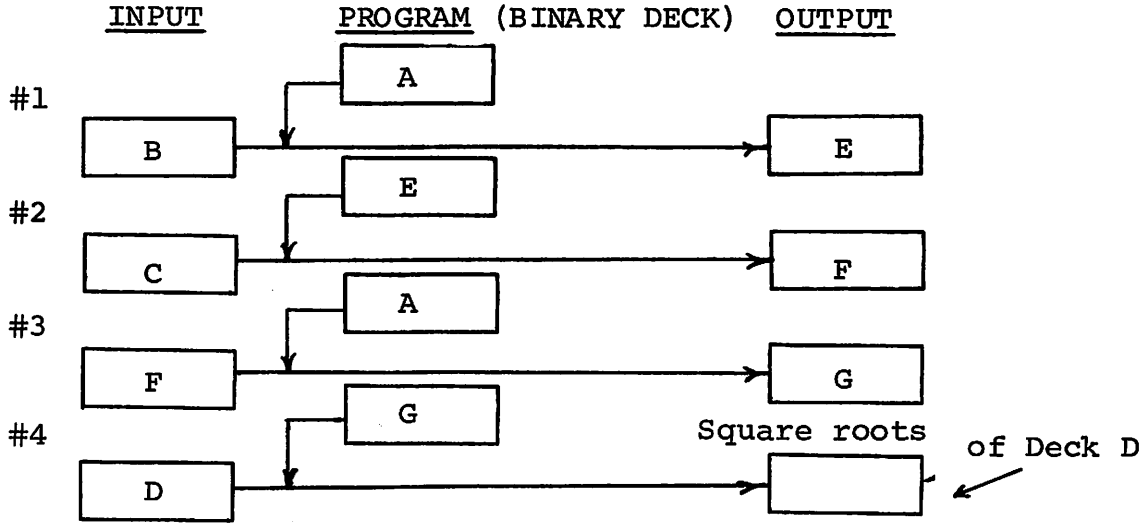
This section presents solutions to the questions in the text. These solutions were written as an aid for the instructor. Most of the solutions are unique and have a concrete answer. However, some questions are design questions and many solutions are possible. We did not attempt to list every subtlety nor did we articulate on all the implications. We apologize for any mistakes you may find, but again, these are meant as an aid.

CHAPTER 1: BACKGROUND

1. (processor, procedure)
A processor is a hardware device that interprets and executes instructions; a procedure is a collection of instructions.
(procedure, program)
There is no difference between procedure and program.
(processor, I/O channel)
An I/O channel is a particular kind of processor, one which interprets I/O instructions.
(multiprocessing, multiprogramming)
In multiprogramming, several programs are started before any one has been executed to completion, regardless of the number of processors; in multiprocessing, several processors work simultaneously on the contents of a single common core memory, regardless of the number of programs in the core.
(open subroutine, closed subroutine)
An open subroutine is an assembly (compile) time phenomenon in which a block of code is inserted into the program; a closed subroutine is at execution time, a separate block of code to which control is transferred.

2. Any set of bits gains meaning only when it is used. The programmer in effect tells the processor how each group of bits is to be interpreted. Once the processor starts execution, the contents of the location counter are used to indicate the location of the next instruction. The address field of the instruction may refer to a location that contains data. The same location may be interpreted at one time as an instruction and at another as data.

3.



4. a. Software
   b. Hardware
   c. Software
   d. Software
   e. Hardware
   f. Hardware
   g. Software
   h. Software*
   i. Software
   j. Hardware

   *A logical unit, although it may sometimes apply to hardware devices, such as card reader or disc.

5. a. The user is given the flexibility of using an available device at run time rather than waiting for a particular unit chosen when the system programs were written. In a similar way, reconfiguration of the system does not require the rewriting of programs to accommodate new device numbers. If a new printer is added to the system, for instance, anyone may use it by changing one DD card.

   b. A table must associate a data set name with the parameters on the card. However, while a name may be defined in several job steps, its entry cannot be changed since cards like

```
// SYSLIN    DD    FROM    STEP1.SYSLIN
```

   refer to definitions made in other job steps. So each name must be associated with a job step name as well, and the table can be erased at the end of each job.

   c. The accounting information from each JOB card must be kept in a permanent (not lost after each job) table. The parameter of each EXEC card must be saved until the end of the job step it initiates.

   d. Control is transferred to INIT/TERM when:

      i) A DD * card is encountered (the data cards are read in under control of the program being executed, not the RDR/INT until the /* is encountered.

      ii) The EXEC card for the next job step is encountered. (The information on this card must be saved to be used when its job step is executed.)

iii) The JOB card for the next job is encountered. (This must also be saved until the job step is completed.)

e. The accounting data base is kept intact.
The DD data base is kept intact.
The EXEC data base is cleared of all information pertaining to the job step just completed. If data for the next job step has been stored (see part d.(ii) above), this will be kept.

f. The accounting data base is kept intact.
The DD data base is cleared.
The EXEC data base is cleared.

g. RDR/INT:

Card format errors (reject job)
Card sequence errors such as JOB followed by DD (reject job)
References to nonexistent programs or data sets (reject job)

INIT/TERM:

Insufficient specification of data sets (use default data sets if possible, or reject job)
Fatal error in program (reject job)

6. a. The PRIORITY of each job would have to be kept, probably in the accounting tables, as it will be used for billing purposes anyway. The queue must also be designed so that it will accommodate all data from DD and EXEC cards associated with the proper job and free the entries for jobs already run.

b. Data cards must be read and saved in the queue now, as they will be physically inaccessible when the program is executed.

c. A job that will take three hours should not be allowed to hold up the execution of short (2-minute) jobs (unless the programmer is willing to pay), for instance. The programmer is allowed to lower his cost by waiting longer, or get almost immediate results if he needs them by paying for the privilege.

1. a-1) OP1 = -26742
OP2 = 26924
OP3 = -31891

OP2 > OP1 > OP3

a-2) OP1 = +979
OP2 = +692
OP3 = +336

OP1 > OP2 > OP3

b-1) OP1 = 20214
OP2 = 24817
OP3 = - 2574

OP2 > OP1 > OP3

b-2) OP1 = + 6
OP2 = - 1
OP3 = 12

OP3 > OP1 > OP2

2. 1) a. 0000 0101 0010 1100

b. (1024+256+32+8+4) = 1324

c. +52

d. BALR 2,12

2) a. 0100 0101 0010 1100

b. (16384+1024+256+32+8+4) = 17768

c. +452

d. Not complete. X'45' is the op-code for BAL, a four-byte instruction.

3) a. 0100 0101 0010 1000 0011 0110 0111 1101

b. (1,393,741,824+67,108,864+16,777,216+2,097,152
+524,288+8192+4096+1024+512+64+32+16+8+4+1)
= 1,480,263,295

c. - 4,528,367

d. BAL 2,871(8,3)

(4) a. 0101 1001 0001 0100 1001 0111 0011 1100

b. (1,393,741,824+348,435,456+174,217,728
+16,777,216+1,048,576+262,144+32,768
+4096+1024+512+256+32+16+8+4
= 1,934,220,800

c. + 5,914,973

d. C 1,1852(4,9)

3. a. If the NEXT field is eliminated, then a location counter (a register, which will contain the address of the next instruction to be executed) can serve the same function as the NEXT field. Usually the next instruction will be in the next location in storage. Thus, each time an instruction is executed, the location counter will be incremented by the length of that instruction. To allow nonsequential operation (loops, decisions, etc.), we need a branch instruction that will load a new value into the location counter.

b. To eliminate the RESULT field, leave the result in the location of OPERAND 1 (or OPERAND 2). Now an instruction is needed to move data in core, as OPERAND 1 will be written over by the result, and a copy must be saved if the data will be needed later.

c. An operand field can be eliminated by assuring that one operand is in the accumulator. An instruction is now needed to load the accumulator with an operand from core.

d. If 24 bits are needed to specify an address (forget base and index registers for the time being), a one-address instruction requires 32 bits (8 bits for the op-code and 24 for the address), while a four-address instruction takes 104 bits. There will be more instructions in a program due to the addition of new functions (load, store, branch), but total program length will be shorter.

The use of an accumulator (usually high-speed logic) is slightly faster than executing an instruction requiring several core memory accesses. However, the programmer must keep track of the contents of the accumulator since it changes so often.

4. a. Intermediate results may be left in one register while operations are done in another. Flexible indexed methods may be used by the programmer. Also, several base registers may be used to address beyond the 4096 byte displacement addressing capability. Operations may take place between registers (which is usually faster than accessing memory).

   b. If used as base or index register, register 0 indicates the absence of a base or index register (i.e., the contents are assumed to be 0). In the BCR and other RR form branch instructions, specifying 0 as the register containing the branch address causes no branch to occur. However, results may be stored into register 0, and arithmetic instructions can operate on the contents of register 0.

5. a. USING is purely an assembly time instruction. It is used to make an entry in the base register table giving the assembler the information it needs to create base register/displacement addresses from symbolic addresses. No machine code is produced from a USING instruction.

   BALR is an executable machine instruction. As such, it is translated by the assembler as is any other machine instruction producing two bytes of machine code. It provides no information to the assembler (unless it has a label, in which case the label is entered into the symbol table).

   b.

```
0   BALR    15,0
2   LR      10,15
4   LH      1,8(0,10)
8   BCR     15,14
10  X'0001'
12  X'0002'
14  X'0003'
```

REG1 contains X'0000 0001'

Assembler is told register 10 will contain relative value 4. At execution time it actually contains relative value 2.

```
0   BALR    15,0
2   LA      10,16(0,15)
6   LH      1,0(0,10)
12  X'0000 0001'
16  X'0000 0002'
20  X'0000 0003'
```

REG1 contains X'0000 0002'

The address of DATA2 is incorrectly loaded into 10 (as 18 instead of 16) because the assembler was told that 15 contains 0 when in fact it is loaded with a 2. The LH, using base register 10, picks up the last halfword of DATA 2.

6. a. No. The first operand in a multiply instruction must be an even register.

   b. Maybe. The dividend is a 64-bit fixed point number in registers 2 and 3. If the contents of register 2 equal 0, then the result will be correct.

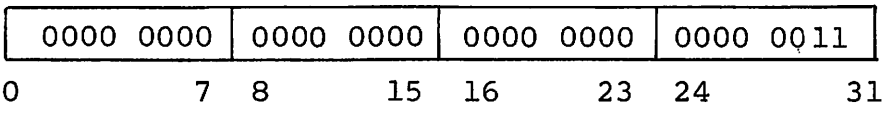
7. a. INDEX EQU 5 is a pseudo-op that causes the assembler to substitute a '5' for every occurrence of INDEX in the program. INDEX DC F'5' is a pseudo-op that causes the assembler to output a 5 in a location named INDEX. This 5 is in a location at execution time; EQU 5 is not around at execution time.

   b. CR treats each operand as a 32-bit signed fixed point number. CLR compares both operands, bit by bit, left to right, treating all bits the same.

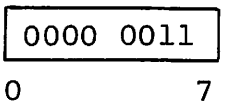
8. a.

```
        LA    3,=A(XYZ)   Address of location of address of
                            XYZ (i.e., address of literal
                            A(XYZ)
        LR    3,3         No change
        L     3,=F'5'     X'0000 0005'
XYZ     LCR   3,3         X'FFFF FFFB'
        LNR   3,3         No change
```

   b. LR 3,5 is next. The CLI instruction compares one byte at the address specified by the first operand with the one byte immediate operand. The literal =F'3' creates a storage location.

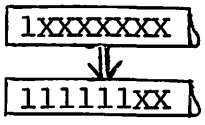


The first byte of this is compared with the immediate operand:

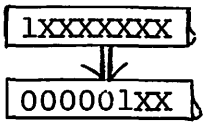


and obviously no match is made.

9. a. If the highest-order bit in register 0 is 1, SRDA 0,5 will treat it as a sign bit and propogate it:



SRDL 0,5 will shift and supply zeroes:



b. SLL 1,1 will shift the register's contents left one place. LA 1,0(1,1) will effectively add the register's contents to itself and then add 0 to the register.

Therefore, these will execute differently unless the high order 9 bits are 0.

c. In an SS format instruction (such as MVC) the length field of the machine code ranges from 0(B'0000') to 15(B'1111') representing lengths of 1 to 16 bytes. The assembler subtracts 1 from the length specified in an instruction unless the length is specified as 0, in which case it remains 0. Therefore, both instructions will produce machine code with a length field B'0000', which will move one character.

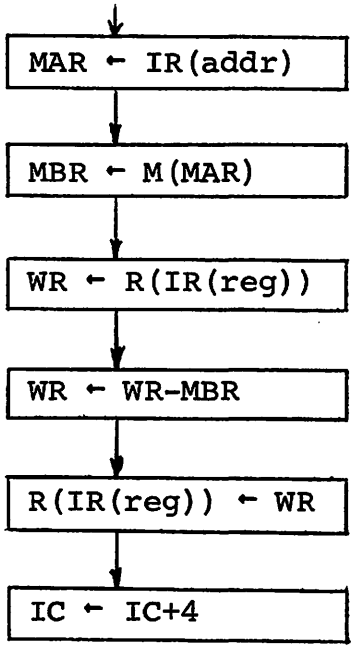
d. LA 1,0(1) will zero the high-order eight bits of register 1. SRA 1,0 will set the condition code.

e. MVC moves bytes one at a time, left to right, from the second operand location to the first operand location.

```
MVC   STOMP+1(8),STOMP   C'ØØØØØØØØØ'
MVC   STOMP(8),STOMP+1   C'ERASUREØØ'
```

10. No. Load, for instance, does neither since checking incurs additional CPU time. The condition code is a record of the result of certain operations (e.g., subtract, test I/O, comparison) that can later be used to determine the effect of a conditional branch. It can save information about a number (+,-,0), about the result of a logical operation, and the status of an I/O process.

11. a. S (subtract, RX form)



Read data from memory

Copy data from designated register into working register

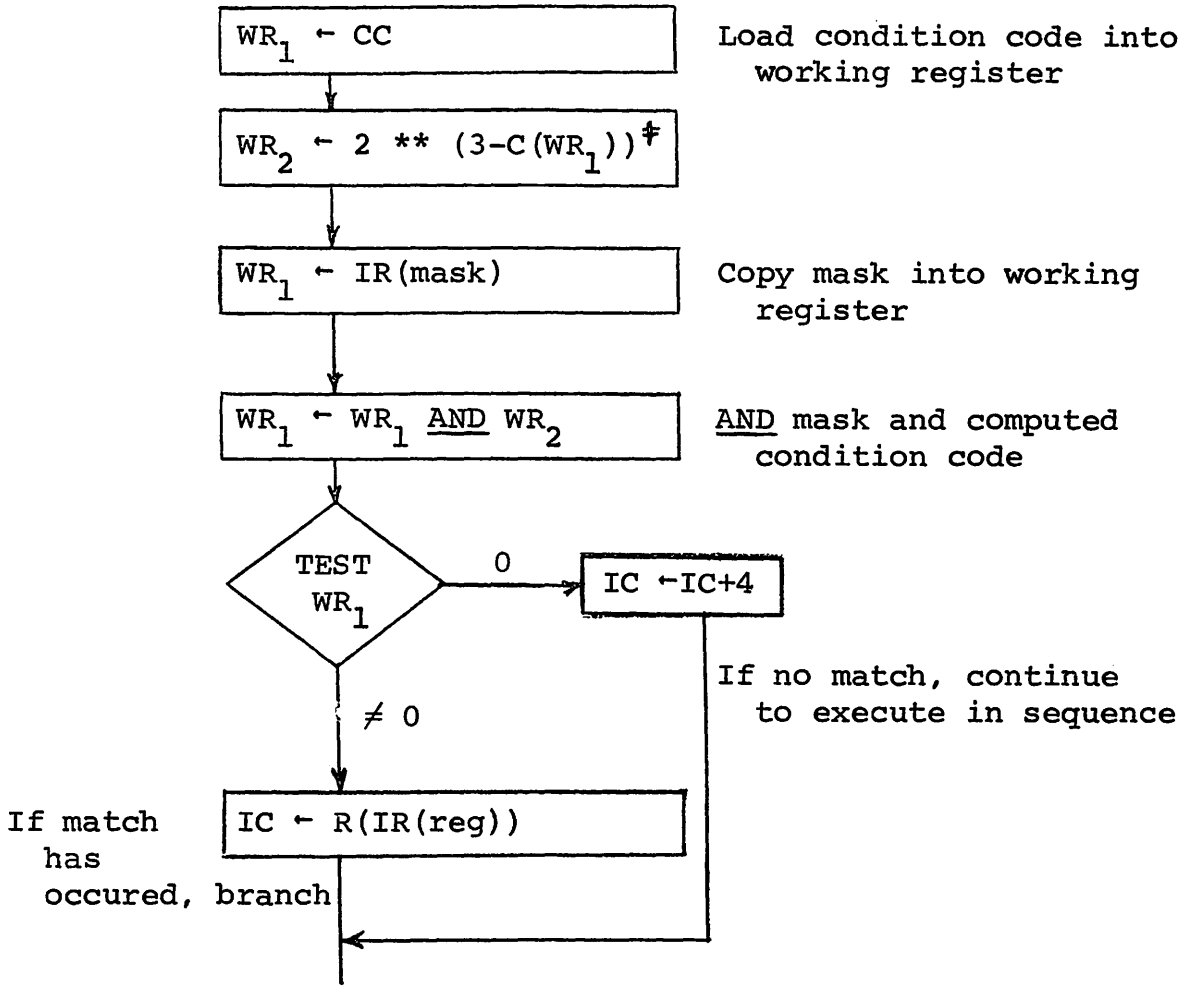
Perform subtraction

Put result into designated register

Increment instruction counter

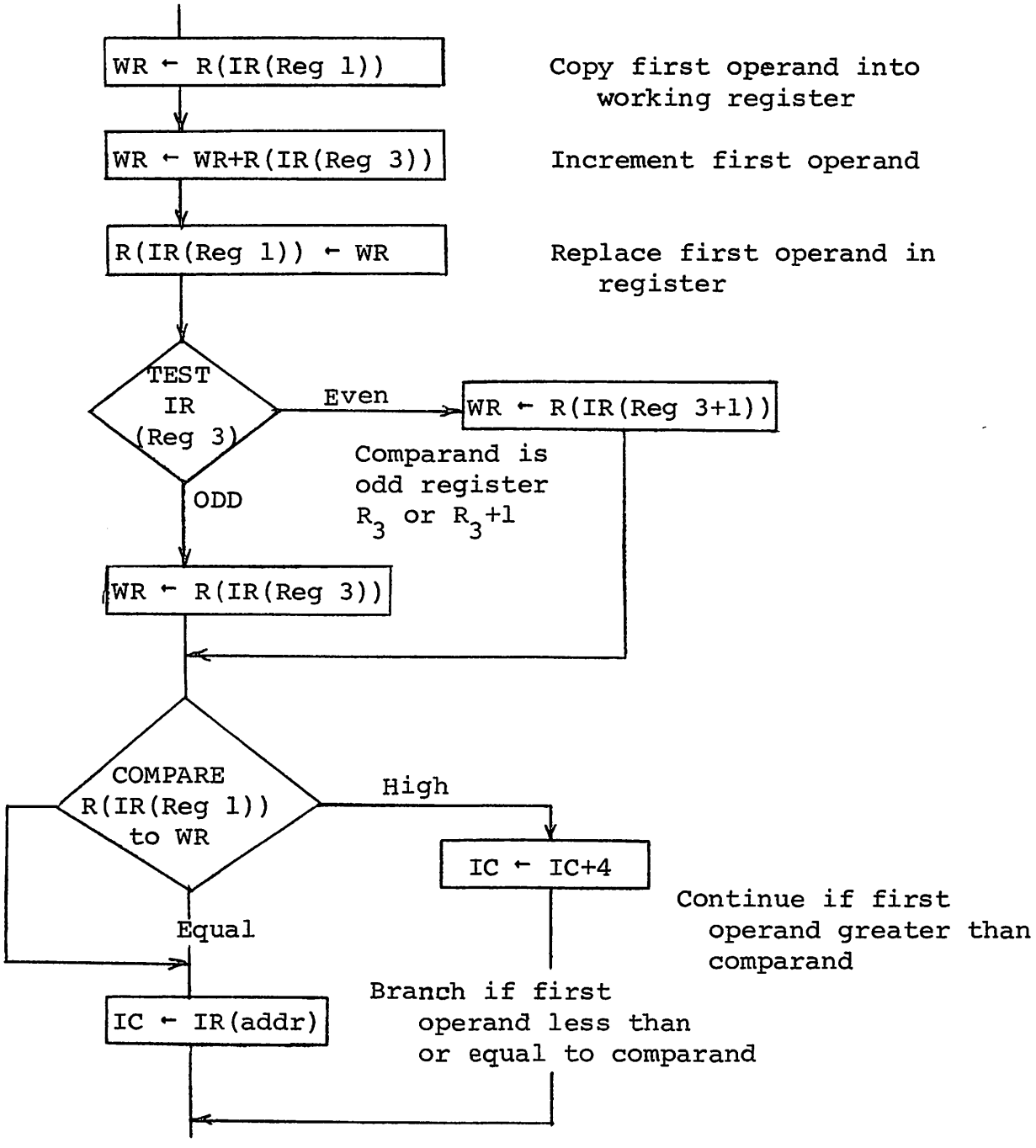
NOTE: No guarantee is made that any of these micro flowcharts agree with IBM's method. They are, however, compatable with their descriptions, and serve as an illustration of how each might be accomplished.

b. BCR (Branch on Condition, RR form)

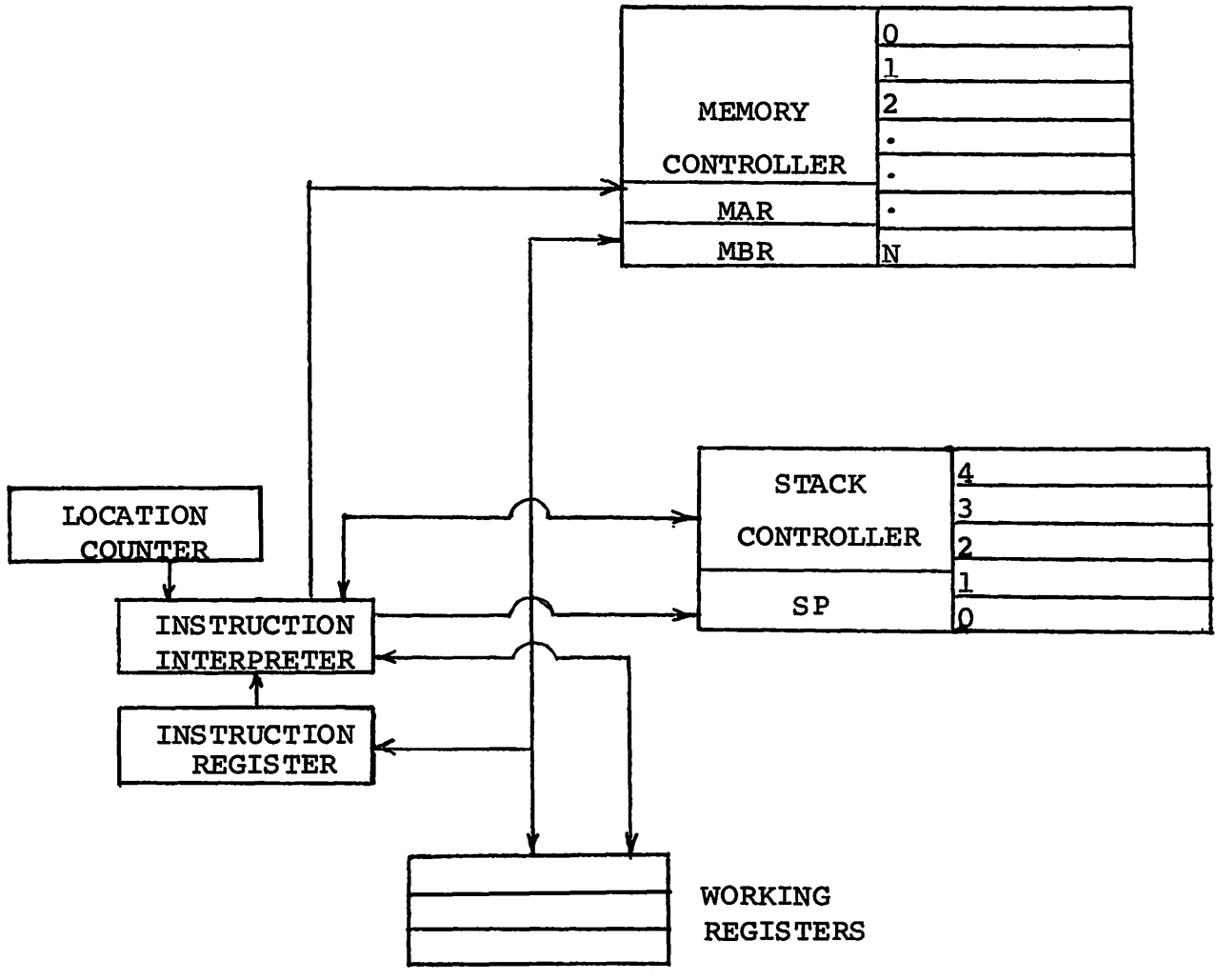


‡Shift a bit (4-C(WR_1)) places in from the right side of WR_2.

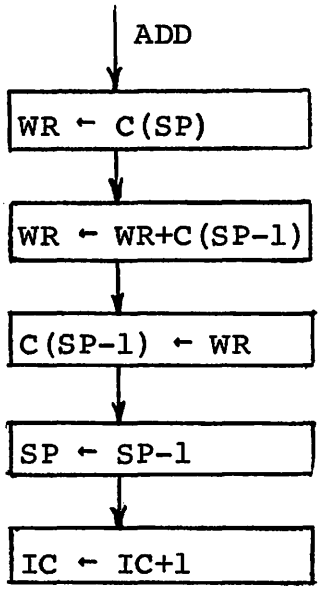
c. BXLE (Branch on Index Less Than or Equal, RS form)



12. a.


MEMORY
CONTROLLER
MAR
MBR
0
1
2
N
STACK
CONTROLLER
SP
4
3
2
1
0
LOCATION
COUNTER
INSTRUCTION
INTERPRETER
INSTRUCTION
REGISTER
WORKING
REGISTERS

b.



Put first operand in working register

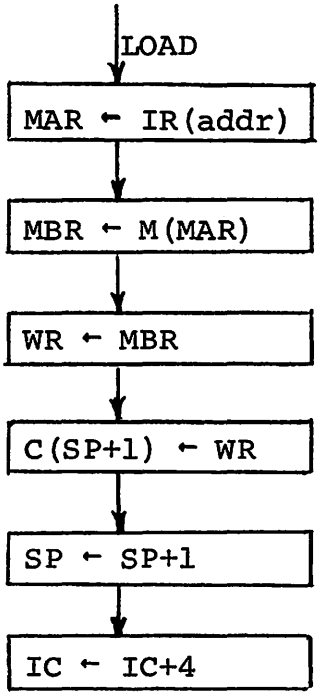
Add second operand to working register

Put result in stack

Decrement stack pointer

Increment location counter

c.



Put address in MAR

Read value from memory

Put value in working register

Put value in stack

Increment stack pointer

Increment location counter

13. a.

Instruction	Contents of Register 1	Register 2	Location A
3	5	?	0
4	10	?	0
5	10	7	0
6	10	14	0
7	24	14	0
8	23	14	0
9	23	14	23

b.

```
3)    LM    1,2,B
4)    AR    1,2
5)    AR    1,1
6)    SH    1,=H'1'
7)    ST    1,A
8)    --    --
9)    --    --
```

Note that an additional 2 bytes was saved by using a halfword literal rather than a fullword literal.

14. See earlier section of this teacher's manual for answers to five questions about a UNIVAC 1108 computer.

15. (Assuming there is a grader)

```
STUDENT   START   0
          ENTRY   STUDENTN
          USING   *,15
          STM     2,3,DATA      Store R2 and R3 in
                                data area
          SR      1,1
          LA      9,DATA        These instructions
          L       10,=F'1'      initialize registers
          L       11,=F'7'      for the BXLE instruc-
                                tion
LOOP      CLI     0(9),C'
          BNE     CHECK
          A       1,=F'1'       increment R1 if
                                comparison true
CHECK     BXLE    9,10,LOOP
          BR      14
DATA      DS      2F            contents of R2 and
                                R3 put here
          LTORG
STUDENTN  EQU     *
          END
```

CHAPTER 3: ASSEMBLERS

1. Basically, the fact that symbols may be referenced before they are defined, such as branches to labels encountered later in the program and symbols defined in EQU and DC statements at the end of the program.

2. Forward references of EQU's could be handled if we had an additional pass.

   Another feature that a three-pass assembler might offer is concerned with optimizing the code. The extra pass could scan the code for such things as multiplication by 2 and then substitute a shift instruction. The pass could look for unnecessary store instructions where it would have been better to leave results in registers as in the following:

```
L     1,ALL                    L     1,ALL
A     1,TEN                    A     1,TEN
ST    1,TEMP      extra
L     1,JOHN      pass  ->     L     2,JOHN
S     1,FOUR                   S     2,FOUR
A     1,TEMP                   AR    1,2
```

   The converted program above is one instruction shortened and has an AR instruction instead of an A, which is slower. Many of the optimization techniques of Chapter 8 may apply here.

   Another feature is that if the extra pass were the first pass, we could test for certain conditions. If the conditions were not satisfied, the assembler would not assemble parts of the program. For example, the user may wish to write a general (but large) program for handling a wide range of data. If he wishes to assemble the program for a smaller range of data, he could place a parameter in his source code, which would be tested and would result in assembling only the parts of the program pertaining to the smaller range of data.

   Other features that an extra pass offers are concerned with macros. (see chapter 4)

   Lastly, an extra pass would use more computer time, making the manufacturers very happy.

3. a. 2
   b. 1,2
   c. 1
   d. 1,2
   e. 1
   f. 2
   g. 1,2
   h. 1,2
   i. 1,2
   j. 2
   k. 2

4. a. Yes.
   b. Yes.
   c. Yes, but each symbol must be defined by an EQU before it is used, and any symbols in operand field must be defined before it is encountered.
   d. Same as (c) above.
   e. Yes.
   f. Yes, left up to the loader.
   g. Yes.
   h. Not at all. (Unless a special loader was designed that could handle this in the loading process)

5. a. Yes. All base registers and displacements must be specified (no symbolic memory reference). This is roughly equivalent to writing in machine languate with mnemonic op-codes and decimal operands.

   b. Forces the location counter to the next double word boundary.

6. During pass 1, each literal encountered will be entered in the literal table. More information is needed, however. Each time an LTORG is encountered, each literal collected since the last LTORG is processed and assigned an address as if it were a DC statement. A one-bit column could be added to the literal table; each literal would flagged when it is entered into a literal pool, and upon encountering an LTORG, all unflagged literals would be processed. The literal pool must actually be formed during pass 1 since the location counter will have to be incremented to include the storage.

Pass 2 could then repeat all the functions of pass 1, creating the machine code for the literal pool, in addition.

Processing of LTORG could have been completely done in Pass 1.

7. a. Although code may not be produced during pass 1, the assembler must have enough information to increment the location counter. The length of the operand of a DC pseudo-op must therefore be computable when first encountered. Thus all symbols must be defined (or self-defined).

   b. The symbol in the label field of an EQU must have a value at the beginning of pass 2. The value must then be assigned when the EQU is just encountered.

   c. No machine code is produced until pass 2. There is no intrinsic difference between the bits representing constants and those representing instructions.

8. The op-codes would have to be added to the machine-op table, and for the purpose of pass 1, all that would be needed is their length for incrementing the location counter.

   For pass 2, rather than just the eight bits of the op-code, the machine-op table would also have to supply the four-bit mask value, and to indicate the special case that all extended mnemonics represent.

   Alternatively, during pass 1 extended mnemonics could be included in the pseudo-op table and replaced with the equivalent BRANCH assembler statement (mask supplied) on the collate tape. Pass 2 would then proceed as always. This involves somewhat more handling but is also somewhat "cleaner" in that it leaves machine-op processing as it was.

9. a.

SYMBOL TABLE

	Value	Length	A/R
SIMPLE	0	1	R
LOOP	2	4	R
R/	1	1	A
TWO	24	4	R
FOUR	28	4	R

LITERAL TABLE

Empty

BASE REGISTER TABLE

	Contents
15	2

CODE

0	BALR	15,0
2	L	1,22(0,15)
6	A	1,22(0,15)
10	ST	1,26(0,15)
14	CLI	29(15),4
18	BC	7,0(0,15)
22	BCR	15,14
24	X'0002'	
28-31	Reserved storage	

*b.

SYMBOL TABLE

	Value	Length	A/R
SAE	488	1	R
ARCHON	1	1	A
DEPUTY	2	1	A
TREAS	3	1	A
BACK	EXT		
BEACON	494	2	R
BETA	548	6	R
STOMP	5	1	A
HOLE	592	8	R
POINT	592	4	R
FOOTBL	EXT		

LITERAL TABLE

	Value	Length	A/R
A(BACK)	576	4	R
H'43'	588	2	R
A(BEACON)	580	4	R
F'482'	584	4	R
F'482'	640	4	R

(Literal pool formed)

BASE REGISTER TABLE

	Base Register	Contents
After STM #6	3	486
After STM #8	3	486
	1	548
	2	4644
After STM #22	3	486
	2	4644

Assembled Code

```
              488       BALR      2,0
              490       LM        1,6,106(3)
              494       CR        2,1
              496       BC        13,38(0,1)
              500       LA        7,28(0,1)
              504       CLI       44(1),X'90'
              508       BCR       15,6
              510       X'0040'
              512       X'40'
              513       X'40'
              514       X'40
              515       X'0000 0000 0000 0000'
              523       Skipped for alignment
              524       X'0000 0186'
              528       X'0000 0186'
              532       X'0000 0186'
```

```
536       X'0000 0186'
540       X'0000 0186'
544       X'0000 0186'
548       MVC         64(4,1),40(1)
554       MVC         60(4,1),40(1)
560       L           9,94(0,3)
564       NR          9,3
566       ST          9,98(0,3)
570       BC          15,130(0,3)
574       Skipped for alignment
576       A(BACK)
580       A(BEACON)
584       X'0000 0188'
588       X'001D'
590       Skipped for alignment
592       X'E6C9 D540'
596       A(BETA)
600       A(BETA)
604       A(BETA)
608       A(BETA)
612       A(BETA)
616       V(FOOTBL)
620       CLC         102(4,3),154(3)
626       LA          1,1(0,0)
630       CVB         2,10(6,0)
634       STC         3,106(9,3)
638       BCR         15,14
640       X'0000 0188'
```

10. a. Both passes will be affected. The symbol table construction will be slightly different, and address formation must change to handle DSECT symbols.

    b. On encountering a DSECT statement, the location counter will be set to 0 and the DSECT label will be put in the symbol table with a flag. Subsequent labels (until a DSECT label) will be stored unflagged.

    In pass 2, on encountering a USING statement the symbol in the address field is checked to see whether it is a DSECT label. If it is flagged, all symbols in the symbol table until the next DSECT label will be flagged and the number of the base register added to the symbol table listing for each, but no entry will be made in the base register table. On forming addresses from a symbolic reference, the symbol will be checked for a flag; if there is one, the base register from the symbol table will be used. Otherwise, the usual base register from the base register table will be used.

    c. Two new listings must be included in the symbol table: a "DSECT flag' column and a "DSECT base register" column.

11. Solution left to instructor.

12. a. 1) Searching is the process of locating data that is to be retrieved from a table.

    2) Sorting involves ordering the contents of a table in order to facilitate searching.

    3) Hashing is a table maintenance/access method in which sorting and searching are not used. Addresses are calculated on the basis of the key itself and can therefore be recalculated for access.

    b. Sorting is the arrangement of a table; searching is retrieving a datum. Sorting is in certain search techniques (e.g., in order to use a binary search, the table must be ordered (sorted) first).

13. (1) a.

	Pass 1	Pass 2	Pass 3	Pass 4	Pass 5	Pass 6	Pass 7	Pass 8	Pass 9	Pass 10	Pass 11
81	52	52	22	22	04	04	04	04	04	04	04
52	~~81~~ 57	~~57~~ 22	52	~~52~~ 04	22	22	22	22	22	22	22
57	~~81~~ 22	57	~~57~~ 04	52	52	42	~~42~~ 32	32	32	32	32
22	81	~~81~~ 04	57	57	57 42	~~52~~ 32	42	42	42	42	~~42~~ 16
95	~~95~~ 04	81	81	~~81~~ 42	~~57~~ 32	~~52~~ 48	48	48	48	~~48~~ 16	42
04	~~95~~ 83	83	~~83~~ 42	~~81~~ 32	~~57~~ 48	52	52	52	~~52~~ 16	48	~~48~~ 14
83	95	~~95~~ 42	~~83~~ 32	~~81~~ 48	57	57	57	~~57~~ 16	52	~~52~~ 14	48
96	~~96~~ 42	~~95~~ 32	~~83~~ 48	~~81~~ 78	78	~~78~~ 65	~~65~~ 16	57	~~57~~ 14	52	52
42	~~96~~ 32	~~95~~ 48	~~83~~ 78	81	~~81~~ 65	~~78~~ 16	65	~~65~~ 14	57	57	57
32	~~96~~ 48	~~95~~ 78	~~83~~ 82	~~82~~ 65	~~81~~ 16	~~78~~ 66	~~66~~ 14	65	65	65	65
48	~~96~~ 78	~~95~~ 82	~~83~~ 65	~~82~~ 16	~~81~~ 66	~~78~~ 14	66	66	66	66	66
78	~~96~~ 82	~~95~~ 65	~~83~~ 16	~~82~~ 66	~~81~~ 14	~~78~~ 77	77	~~77~~ 67	67	67	67
82	~~96~~ 65	~~95~~ 16	~~83~~ 66	~~82~~ 14	~~81~~ 77	78	~~78~~ 67	77	77	77	77
65	~~96~~ 16	~~95~~ 66	~~83~~ 14	~~82~~ 77	81	~~81~~ 67	78	78	78	78	78
16	~~96~~ 66	~~95~~ 14	~~83~~ 77	82	~~82~~ 67	81	81	81	81	81	81
66	~~96~~ 14	~~95~~ 77	83	~~83~~ 67	82	82	82	82	82	82	82
14	~~96~~ 77	~~95~~ 87	~~87~~ 67	83	83	83	83	83	83	83	83
77	~~96~~ 87	~~95~~ 67	87	87	87	87	87	87	87	87	87
87	~~96~~ 67	95	95	95	95	95	95	95	95	95	95
67	96	96	96	96	96	96	96	96	96	96	96

04	04	04	04	04
22	~~22~~ 16	16	14	14
~~32~~ 16	22	~~22~~ 14	16	16
32	~~32~~ 14	22	22	22
~~42~~ 14	32	32	32	32
42	42	42	42	42
48	48	48	48	48
52	52	52	52	52
57	57	57	57	57
65	65	65	65	65
66	66	66	66	66
67	67	67	67	67
77	77	77	77	77
78	78	78	78	78
81	81	81	81	81
82	82	82	82	82
83	83	83	83	83
87	87	87	87	87
95	95	95	95	95
96	96	96	96	96
Pass	**Pass**	**Pass**	**Pass**	**Pass**
12	**13**	**14**	**15**	**16**
				No change -- DONE

(2) a.

81	*48	*04	04	04	04
52	*52	*14	14	14	14
57	57	57	**~~48~~ 42	*~~42~~ 16	16
22	22	22	22	22	22
95	*16	16	16	*~~42~~ 32	32
04	04	*48	**~~57~~ ~~42~~ 48	48	*~~48~~ 42
83	*14	*52	*~~52~~ 32	*42	*48
96	*77	77	**~~77~~ ~~66~~ 65	*~~65~~ 52	52
42	42	42	*57	57	57
32	32	32	*52	*65	65
48	*81	*~~81~~ 66	**77 ~~65~~ 66	66	66
78	78	78	*~~78~~ 67	67	67
82	82	82	*~~82~~ 81	*~~81~~ 78	*~~78~~ 77
65	65	65	*77	77	*78
16	*95	*~~95~~ 67	*78	*81	81
66	66	*~~66~~ 81	*82	82	82
14	*83	83	83	83	83
77	*96	96	96	*~~96~~ 95	*~~75~~ 87
87	87	87	87	87	95
67	67	*95	95	*96	96
	Pass 1 d=10	Pass 2 d=5	Pass 3 d=3	Pass 4 d=2	Pass 5 d=1

(3) a.

	First Distribution	Merge	Second Distribution	Merge
81	0)	81	0) 04	04
52	1) 81	52	1) 14,16	14
57	2) 52,22,42,32,82	22	2) 22	16
22	3) 83	42	3) 32	22
95	4) 04,14	32	4) 42,48	32
04	5) 95,65	82	5) 52,57	42
83	6) 96,16,66	83	6) 65,66,67	48
96	7) 57,77,87,67	04	7) 77,78	52
42	8) 48,78	14	8) 81,82,83,87	57
32	9)	95	9) 95,96	65
48		65		66
78		96		67
82		16		77
65		66		78
16		57		81
66		77		82
14		87		83
77		67		87
87		48		95
67		78		96

(1) b.

424	424	424	424	424	424	424
887	~~887~~ 807	~~807~~ 709	709	~~709~~ 616	~~616~~ 573	~~573~~ 413
807	~~887~~ 709	807	~~807~~ 016	~~709~~ 573	~~616~~ 413	573
709	~~887~~ 882	~~882~~ 616	~~807~~ 573	~~709~~ 413	616	~~616~~ 180
882	~~887~~ 616	~~882~~ 573	~~807~~ 413	~~709~~ 679	~~679~~ 180	616
616	~~887~~ 573	~~882~~ 413	~~807~~ 679	~~709~~ 180	679	~~679~~ 264
573	~~887~~ 413	~~882~~ 679	~~807~~ 180	709	~~709~~ 264	679
413	~~887~~ 679	~~882~~ 180	807	~~807~~ 264	709	709
679	~~887~~ 180	882	~~882~~ 264	807	807	807
180	887	~~887~~ 264	882	882	882	882
975	~~975~~ 264	887	887	887	887	887
264	975	975	975	975	975	975
	Pass 1	Pass 2	Pass 3	Pass 4	Pass 5	Pass 6

~~424~~ 413	413	180	180	180
424	~~424~~ 180	413	264	264
~~573~~ 180	424	~~424~~ 264	413	413
573	~~573~~ 264	424	424	424
~~616~~ 264	573	573	573	573
616	616	616	616	616
679	679	679	679	679
709	709	709	709	709
807	807	807	807	807
882	882	882	882	882
887	887	887	887	887
975	975	975	975	975
Pass 7	Pass 8	Pass 9	Pass 10	Pass 11

No change -- DONE

(2) b.

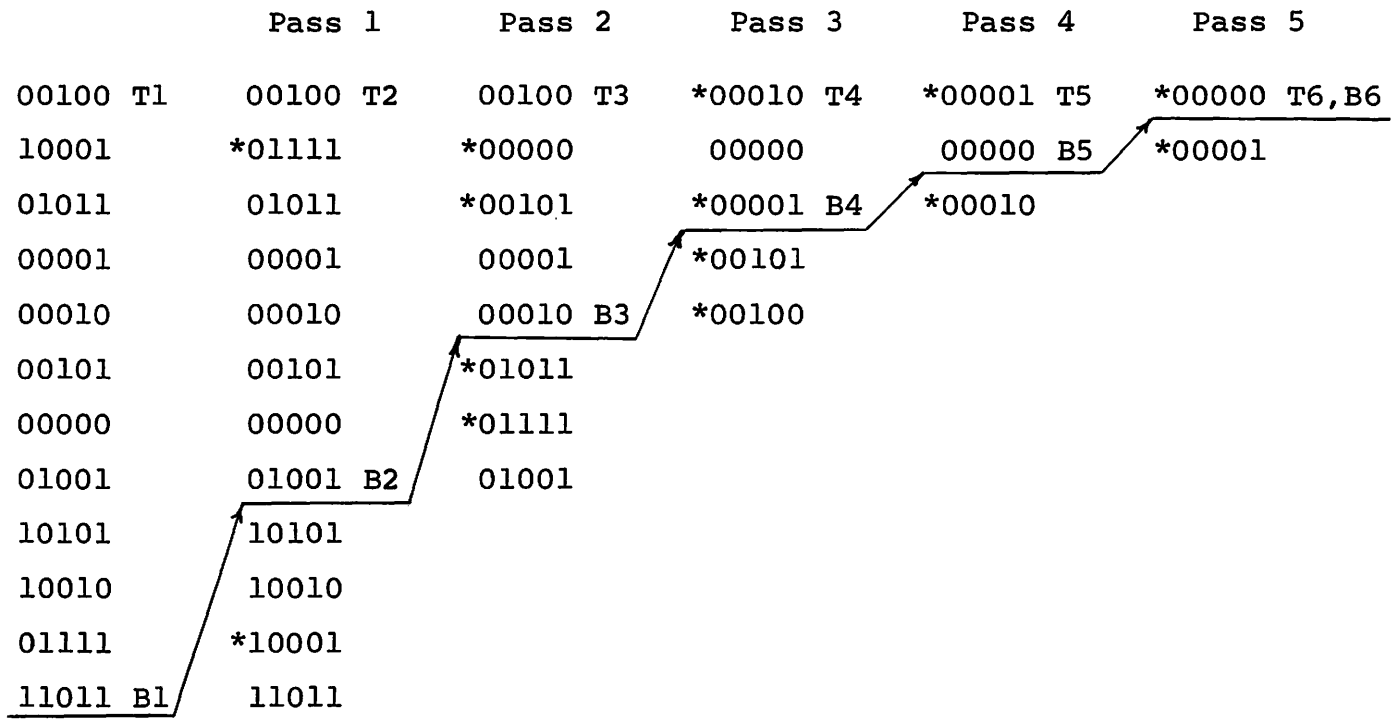
424	424	*180	180	180
887	*413	413	413	*~~413~~ 264
807	*679	*264	264	*413
709	*180	*424	424	424
882	882	882	*~~882~~ 573	573
616	*264	**~~679~~ 616	616	616
573	573	573	**~~882~~ 679	679
413	*887	887	*~~887~~ 709	709
679	*807	*~~807~~ ~~616~~ 679	*882	*~~882~~ 807
180	*709	709	*~~887~~ 807	882
975	975	975	975	*~~975~~ 887
264	*616	*807	*887	*975
	Pass 1 d=6	Pass 2 d=3	Pass 3 d=2	Pass 4 d=1

(3) b.

	First Distribution	Merge	Second Distribution	Merge	Third Distribution	Merge
424	0) 180	180	0) 807,709	807	0)	180
887	1)	882	1) 413,616	709	1) 180	264
807	2) 882	573	2) 424	413	2) 264	413
709	3) 573,413	413	3)	616	3)	424
882	4) 424,264	424	4)	424	4) 413,424	573
616	5) 975	264	5)	264	5) 573	616
573	6) 616	975	6) 264	573	6) 616,679	679
413	7) 887,807	616	7) 573,975,679	975	7) 709	709
679	8)	887	8) 180,882,887	679	8) 807,882,887	807
180	9) 709,679	807	9)	180	9) 975	882
975		709		882		887
264		679		887		975

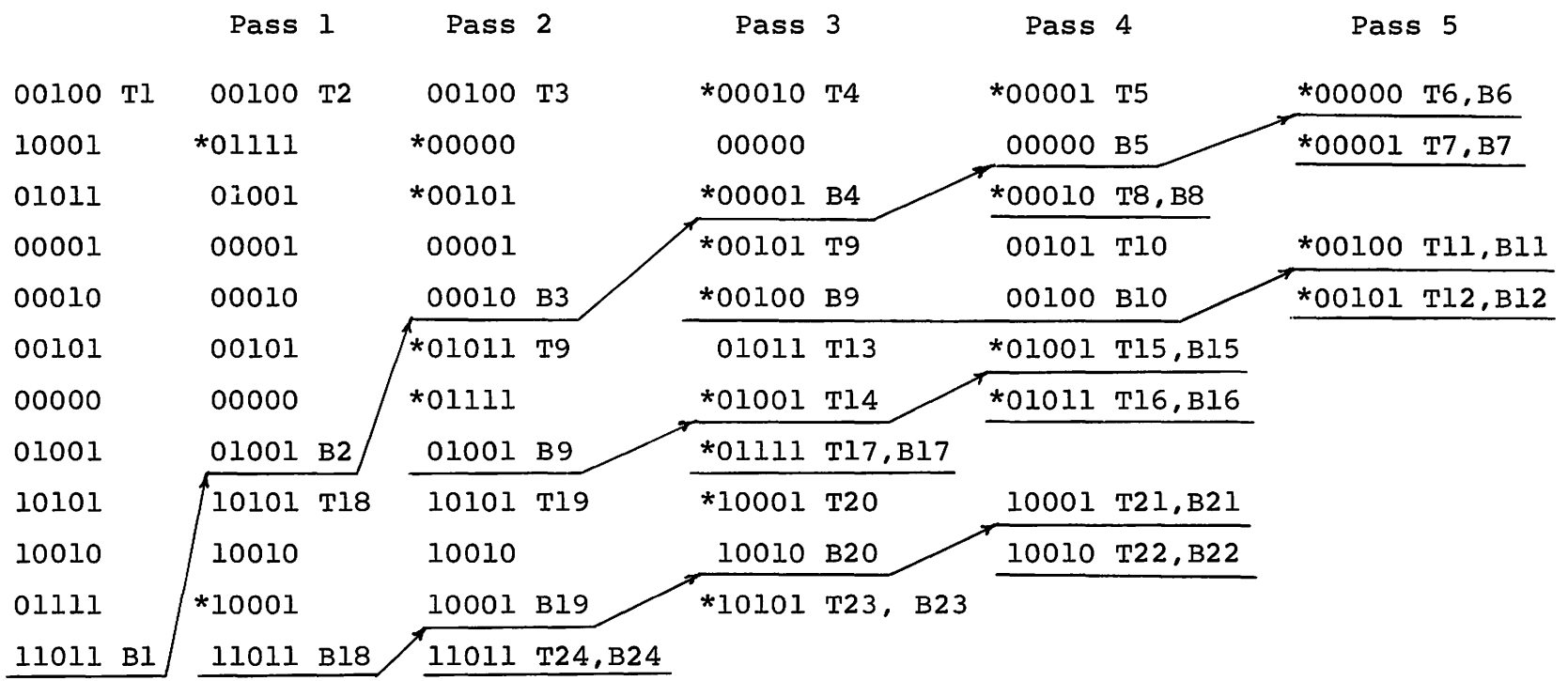
14. 1) 4 probes
2) 3 probes
3) 4 "
4) 2 "
5) 4 "
6) 3 "
7) 4 "
8) 1 "
9) 4 "
10) 3 "
11) 4 "
12) 2 "
13) 4 "
14) 3 "
15) 4 "

15. a.



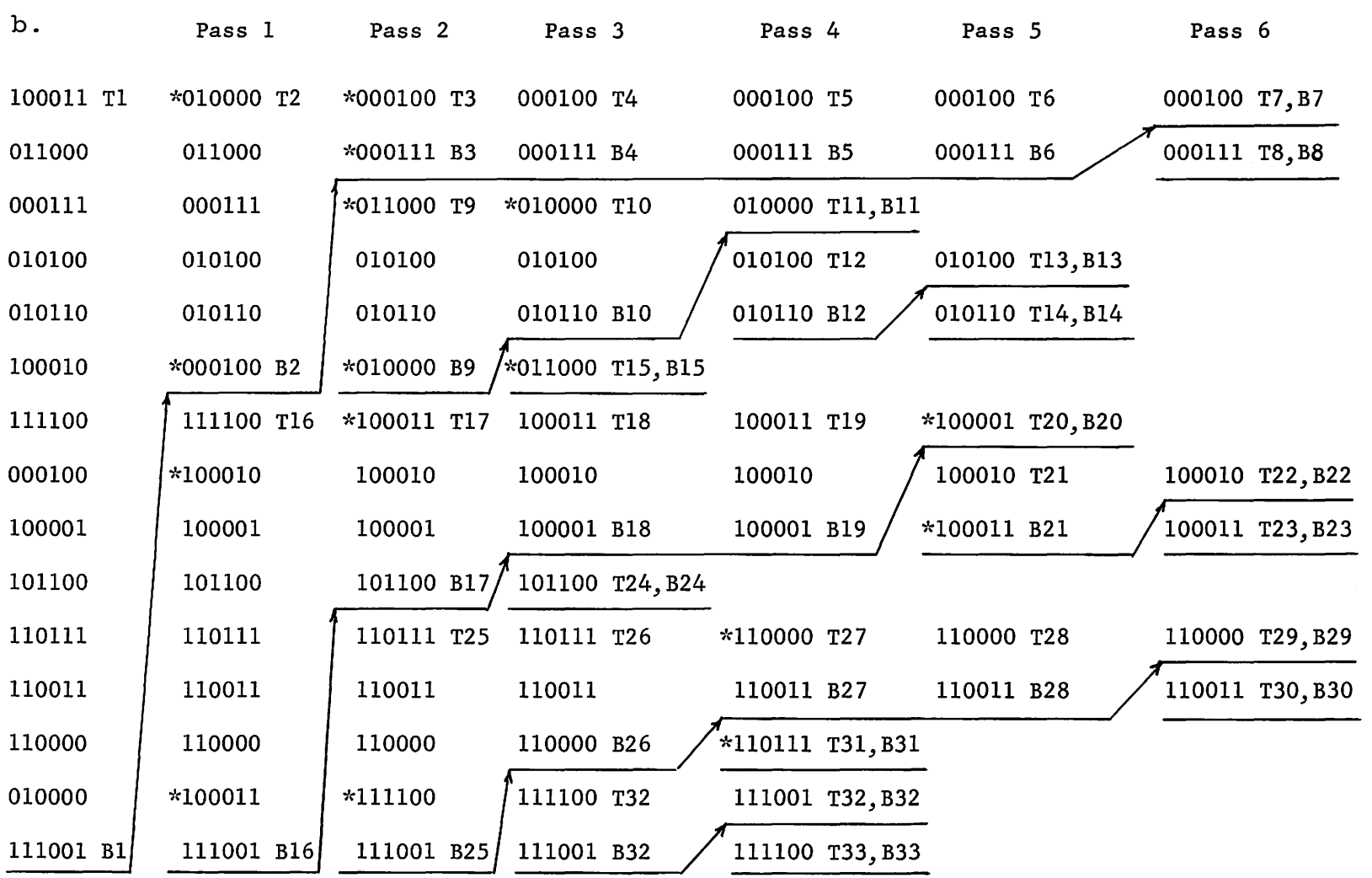
First set of breakpoints

Exchanges indicated by (*)



Final ordering of table

b.


Pass 1
Pass 2
Pass 3
Pass 4
Pass 5
Pass 6
100011 T1
011000
000111
010100
010110
100010
111100
000100
100001
101100
110111
110011
110000
010000
111001 B1
*010000 T2
011000
000111
010100
010110
*000100 B2
111100 T16
*100010
100001
101100
110111
110011
110000
*100011
111001 B16
*000100 T3
*000111 B3
*011000 T9
010100
010110
*010000 B9
*100011 T17
100010
100001
101100 B17
110111 T25
110011
110000
*111100
111001 B25
000100 T4
000111 B4
*010000 T10
010100
010110 B10
*011000 T15,B15
100011 T18
100010
100001 B18
101100 T24,B24
110111 T26
110011
110000 B26
111100 T32
111001 B32
000100 T5
000111 B5
010000 T11,B11
010100 T12
010110 B12
100011 T19
100010
100001 B19
*110000 T27
110011 B27
*110111 T31,B31
111001 T32,B32
111100 T33,B33
000100 T6
000111 B6
010100 T13,B13
010110 T14,B14
*100001 T20,B20
100010 T21
*100011 B21
110000 T28
110011 B28
000100 T7,B7
000111 T8,B8
100010 T22,B22
100011 T23,B23
110000 T29,B29
110011 T30,B30

16.

Attributes \ Sorts	Shell	Radix	Interchange	Radix Exchange
Examines all bits (or characters) in key of an item in each probe	✔		✔	
Requires large amounts of extra storage		✔		
Requires only one pass if table is already sorted on entry to sort			✔	
Is a distributive sort		✔		✔
Requires items to be sorted to be in binary form				✔

17. This is dependent upon the limitations of the machine, such as available core and the potential size of the symbol table. Basically, the problem is that:

    a. Frequent entries will be made so that keeping an ordered table would involve a great deal of overhead (moving existing entries around to make room).

    b. The table may be accessed often, and it is slow to have to search an unordered table.

    If enough core is available, hash coding is the obvious answer because no ordering is needed or used. If the number of symbols is small, then the best and easiest method of searching would be linear.

18. a. Symbol table < 100: use a comparative sort; with a table this size speed is comparable, but comparative sorting techniques require no extra storage.

Symbol table > 100: use a distributive sort here, as page swapping will have to be done anyway, and the use of extra storage becomes relatively unimportant. The distributive sorts would be faster.

b.

	Case 1 table size < 100 accessed infrequently	Case 2 table size < 100 accessed frequently	Case 3 table size > 100 accessed infrequently	Case 4 table size > 100 accessed frequently
linear search	X			
binary search		X		
hash search			X	
hash with binary bucket search				X
sequential entry	X			
sequential entry with interchange sort				
sequential entry with radix sort				
sequential entry with Shell sort		X		
hash entry			X	
hash entry with bucket sort				X

19. a.

```
22000  MULHERNØ
22008  HARRISØØ
22016
22024  MARYØØØØ
22032
22040
22048
22056  NANGLEØØ
22064  MITCHØØØ
22072  MADNICKØ
22080
22088
22096
22104  DONOVANØ
22112  FREYBERG
22120
22128
22136
22144
22152
```

DONOVAN- $(4+6+5+6+4+1+5+0)_{MOD\ 19} = 13$

MULHERN- $(4+4+3+8+5+9+5+0)_{MOD\ 19} = 0$

FREYBERG $(6+9+5+8+2+5+9+7)_{MOD\ 19} = 13$

MITCH--- $(4+9+3+3+8+9+0+0)_{MOD\ 19} = 8$

HARRIS-- $(8+1+9+9+9+2+0+0)_{MOD\ 19} = 0$

MARY---- $(4+1+9+8+0+0+0+0)_{MOD\ 19} = 3$

MADNICK- $(4+1+4+5+9+3+2+0)_{MOD\ 19} = 9$

NANGLE-- $(5+1+5+7+3+5+0+0)_{MOD\ 19} = 7$

b. The sequential overflow method used here is poor in a crowded table, but worse, if an entry is deleted, the whole process may be disrupted. For instance,

```
129
130   A
131   C
132   E
133
```

If the keys for A, C, and E all resulted in 130, they would be stored this way. If C is deleted, the question of what to do with E arises.

An overflow area separate from the main table could be be established. Compacting is still a problem, however. Another method of computing an address from the key could be used. This, however, requires more computation.

It would be simpler to establish chains, including pointers to the next variable with the same calculated address. Detection now becomes much easier.

20. a. Shell sort separations: 11, 6, 3, 2, 1

ASHTON, BEN
BERMAN, HARRIS
BRUNK, GLEN
DODSON, ORVILLE
DONOVAN, CAROLYN
DONOVAN, MAUREEN
DONOVAN, JAMES
DONOVAN, REBECCA
DONOVAN, MARILYN
DONOVAN, JOHN
FREYBERG, DUTCH
GOODMAN, LEONARD
HAMMER, MICHAEL
JACK, MARTIN
JOHNSON, JERRY
KOHN, NORMAN
MADNICK, STUART
NANGLE, ELLEN
RAMCHANDANI, CHANDER
SINNOT, MARY
ZILLES, STEVEN

b. I = 6

c. Binary search for last name: fast, no extra storage requirement. Backs up to first occurence of last name and performs a linear search for first name: relatively slow, no extra storage requirement.

d. It is simple to implement and for lists of the typical format and size described in the problem, the algorithm is probably a good compromise between size and speed.

e. For large lists of names, the binary search portion is all right, but backing up and then doing a linear search could become inordinately slow if a number of identical last names were in the list.

CHAPTER 4: MACRO LANGUAGE AND THE MACRO PROCESSOR

1. The important thing here is that the input and output are text. Except for macro instructions, such as MACRO, AIF, and MEND, the actual characters are immaterial; thus, almost any format could be accommodated.

2. a. Macro Definition Table

```
 1.    XYZ     &A
 2.    ST      1,&A
 3.    MEND
 4.    MIT     &Z
 5.    MACRO
 6.    &Z      &W
 7.    AR      4,&W
 8.    XYZ     ALL
 9.    MEND
10.    ST      &Z,ALL
11.    MEND
12.    HELLO   &W
13.    AR      4,&W
14.    XYZ     ALL
15.    MEND
```

Macro Name Table

```
1.    XYZ     1
2.    MIT     4
3.    HELLO   12
```

Expanded Assembler Language

```
PROG  START
      USING   *,15
      ST      2,3
      AR      4,YALE
      ST      1,ALL
YALE  EQU     5
ALL   DC      F'3'
      END
```

b. Macro Definition Table

```
1.              EXPO    &EXP
2.              LCLA    &N
3.      &N      SETA    &EXP
4.              AIF     (&N EQ 1).STOP
5.              MR      0,2
6.      &N      SETA    &N-1
7.              EXPO    &N
8.      .STOP   ANOP
9.              MEND
```

Macro Name Table

```
1.      EXPO    1
```

Expanded Assembler Language

```
EXPON   START   0
        USING   0,15
        L       2,BASE
        L       1,BASE
        SR      0,0
        MR      0,2
        MR      0,2
        ST      1,ANS
        BR      14
ANS     DS      F
BASE    DC      F'5'
        END
```

3.

	Must be built or modified in pass 1	Referenced but not modified in pass 1	Built during pass 1; must be saved for pass 2	Not modified during pass 2	Built or modified during pass 2	References to the data base occur in pass 2
Symbol table	✓		✓	✓		✓
Literal table	✓		✓	✓		✓
Base register table					✓	✓
Collate tape	✓		✓	✓		✓
Machine Op Table (MOT) includes MNT	✓			✓		✓
Pseudo-op table		✓		✓		✓
Input deck		✓		✓		✓
Macro Definition Table (MDT)	✓					
Location counter	✓				✓	✓
Macro stack	✓					

4. This was touched upon in Chapter 1 as open versus closed subroutines. In a subroutine, call control is transferred from the calling procedure to another separate procedure. In a macro expansion code is inserted into the program. Subroutine calls occur at execution time, macro expansion at assembly time. A macro for a routine called several times may take a great deal of space but no overhead in transferring to it (i.e., no calling sequence), while a subroutine may take some time in the overhead of entering and exiting a procedure. However, only one copy of the closed subroutine exists.

   The analog of the stack frame is the push down or LIFO (last-in/first-out) stack, which contains intermediate values for recursive routines, in particular, return addresses. This is discussed further in section 8.4 -- Recursion.

5. a. No. The cards of the nested macro are stored in the MDT as are all other cards in the external macro definition.

   b. No.

   c. Yes. The macro will be defined when, after the external macro is called, the internal macro definition is encountered. At this point, entries in the MNT and MDT will be made, and the macro may be called.

   d. No. There will be no entries in the MDT and MNT until after the internal macro, so it cannot be referenced before its definition.

   e. Recursive macros involve no internal macro definitions, but they do require a stack frame. Therefore, no.

   f. Macros calling themselves and producing intermediate values which must be saved.

6. Use of argument supplied labels, such as:

```
            MACRO
            ADD     &LAB,&LOC
&LAB        A       1,&LOC
            MEND
```

presents no problem, but a label such as:

```
            MACRO
            ADD     &LOC
LABEL       A       1,&LOC
            MEND
```

does because the label will be inserted each time the macro is called, resulting in a multiply-defined symbol. The macro processor must generate a new symbol each time the macro is used.

7. a. All macros must be defined before they are referenced.

   b. In general, a one-pass assembler can handle a macro call with conditional macro pseudo-ops. Additional passes may be necessary if very felxible conditions are required (e.g. the example in the question).

   c. Either have additional passes or restrict conditions in order to not allow testing of a variable that hasn't been assigned a value previously.

8. a. (SR)

```
            MACRO
&LAB1       SR      &OP1,&OP2
&LAB1       SET     LC
            BYTE    1B(16,8)
LC          SET     LC+1
            BYTE    &OP1(10,4),&OP2(10,4)
LC          SET     LC+1
            MEND
```

b. This is a difficult problem, but a good one to show the power of a macro capability. One solution is to define macros that can accept statements like L 1,900 and produce the machine code

58	1	0	0	900
op	R1	X2	B2	offset

Other solutions may be even more elaborate and would allow the user to specify a base register. To do this, we must do the following.

To start, a good solution should initialize the LC variable, and since we will need a base register, initialize a BASE variable. Therefore, we may define a START macro:

```
         MACRO
&LBL     START     &LOC
LC       SET       &LOC
&LBL     SET       LC
ORIGIN   SET       LC
BASE     SET       Ø
         MEND
```

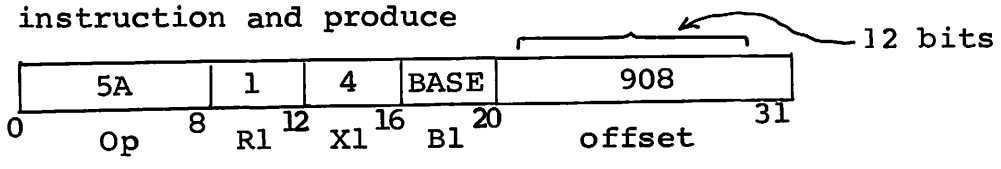
To set the value of the base register, a using macro can be defined as follows.

```
         MACRO
&LBL     USING     &LOC,&REG
BASE     SET       &REG
ORIGIN   SET       &LOC
&LBL     SET       LC
         MEND
```

This will assign a base register and its contents without conditional pseudo-ops; no more than one BASE can be active.

Add Instruction, RX Format

Note: An index register may be specified by the last argument, e.g. an ADD instruction using register 4 as an index would look like A 1,908,4. The macro assembler must accept this ADD instruction and produce



where 1, 4,908 would be binary and BASE would have been defined in a USING card.

The following MACRO definition defines such an add instruction. Note the complications necessary to compute the offset and place it in the appropriate 12 bits.

Add Instruction, RX Format

Note: An index register is specified by the last argument, e.g., an add instruction using register 4 as an index would look like the following: A 1,908,4

```
         MACRO
&LBL     A       &REG,&DATA,&INDEX
&LBL     SET     LC
         BYTE    5A(16,8)
LC       SET     LC+1
         BYTE    &REG(10,4),&INDEX(10,4)
LC       SET     LC+1
         BYTE    BASE(10,4),&DATA/256-ORIGIN/
                    256(10,4)
LC       SET     LC+1
         BYTE    &DATA-ORIGIN(10,8)
LC       SET     LC+1
         MEND
```

This assumes null index argument is evaluated as Ø, and that expression (in last BYTE) is truncated to fit. Parentheses for index specification requires more machinery.

For other op-codes, let us define the macro LOC for computing and storing the offset and the base register.

```
         MACRO
         LOC     &DATA
         BYTE    BASE(10,4),&DATA/256-ORIGIN/
                    256(10,4)
LC       SET     LC+1
         BYTE    &DATA-ORIGIN(10,8)
LC       SET     LC+1
         MEND
```

Now we may define a macro for L instruction.

```
            MACRO
&LBL        L           &REG,&DATA,&INDEX
&LBL        SET         LC
            BYTE        58(16,8)
LC          SET         LC+1
            BYTE        &REG(10,4),&INDEX(10,4)
LC          SET         LC+1
            LOC         &DATA
            MEND
```

9. a. 1) All macro processing occurs in pass 1 of the assembler.

2) LCLA and GBLA will change the definition phase but will have no direct effect on the expansion phase.

SETA will be stored in the MDT with no change to the definition phase. All changes will be made to the expansion phase.

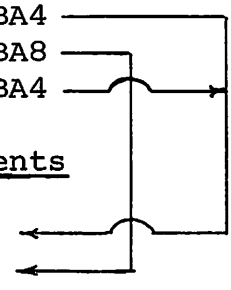
b. There is no real reason to do this, as only one location is needed for each GBLA symbol or each occurrence of an LCLA symbol. Once a value is updated, the old value has no significance, and it is unnecessary to use the stack to hold the other values.

A Macro Symbol Table (MST) could be established instead to associate each symbol with its current value. This is much more direct. A SETA statement would update its symbol in the MST.

Macro Symbol Table

Symbol	Macro	GBL/LCL	Value Pointer
&A	M1	G	0000 1BA4
&A	M2	L	0000 1BA8
&A	M3	G	0000 1BA4

Core Location	Contents
0000 1BA4	5
0000 1BA8	2

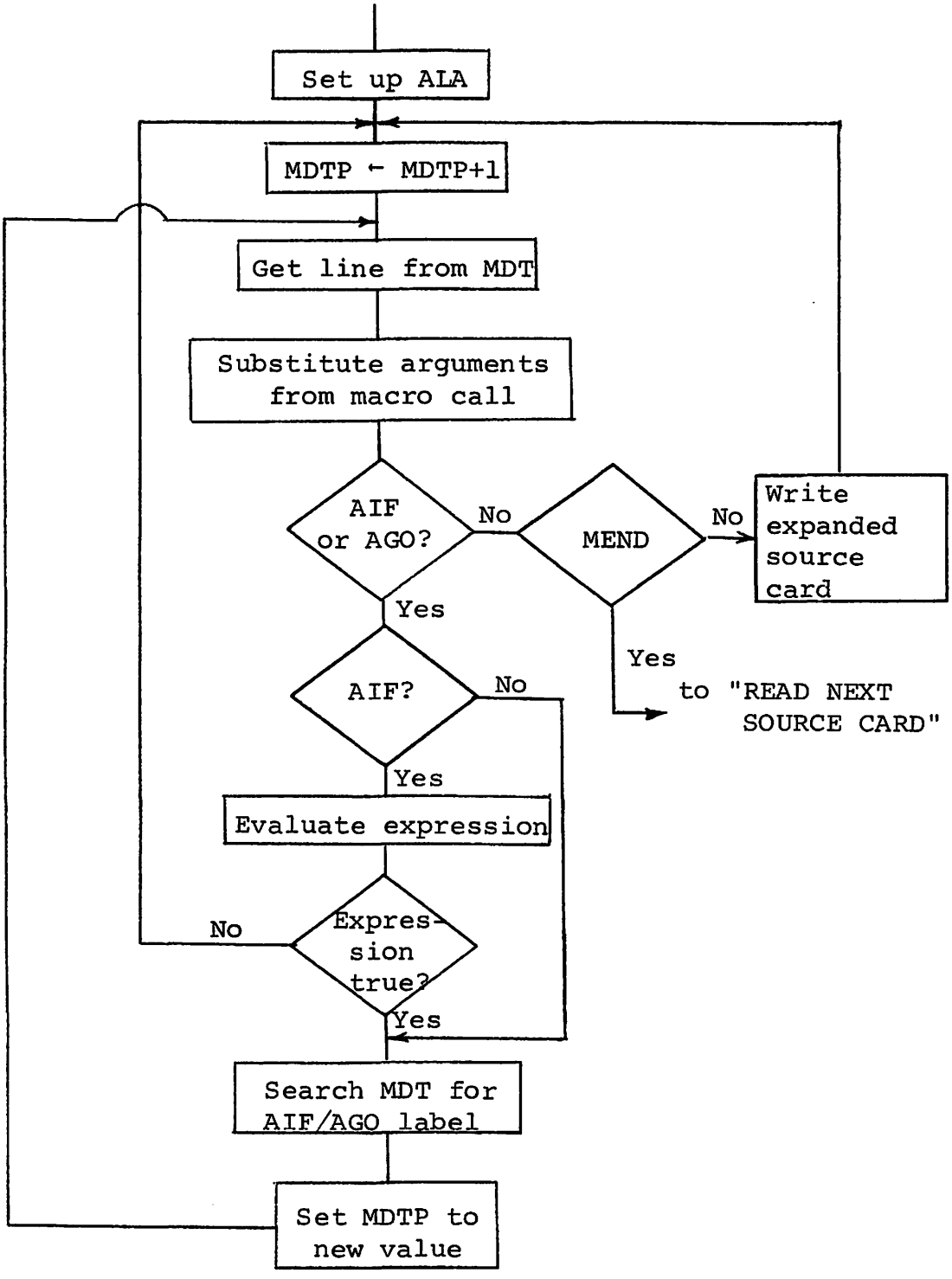


c. Search the MST for the symbol in the label field of the SETA statement. Then find the correct symbol by searching the macro column as well.for the macro in which the SETA occurred. Then modify the location to which the value pointer points to reflect the SETA statement. No difference between handling of local and global symbols.

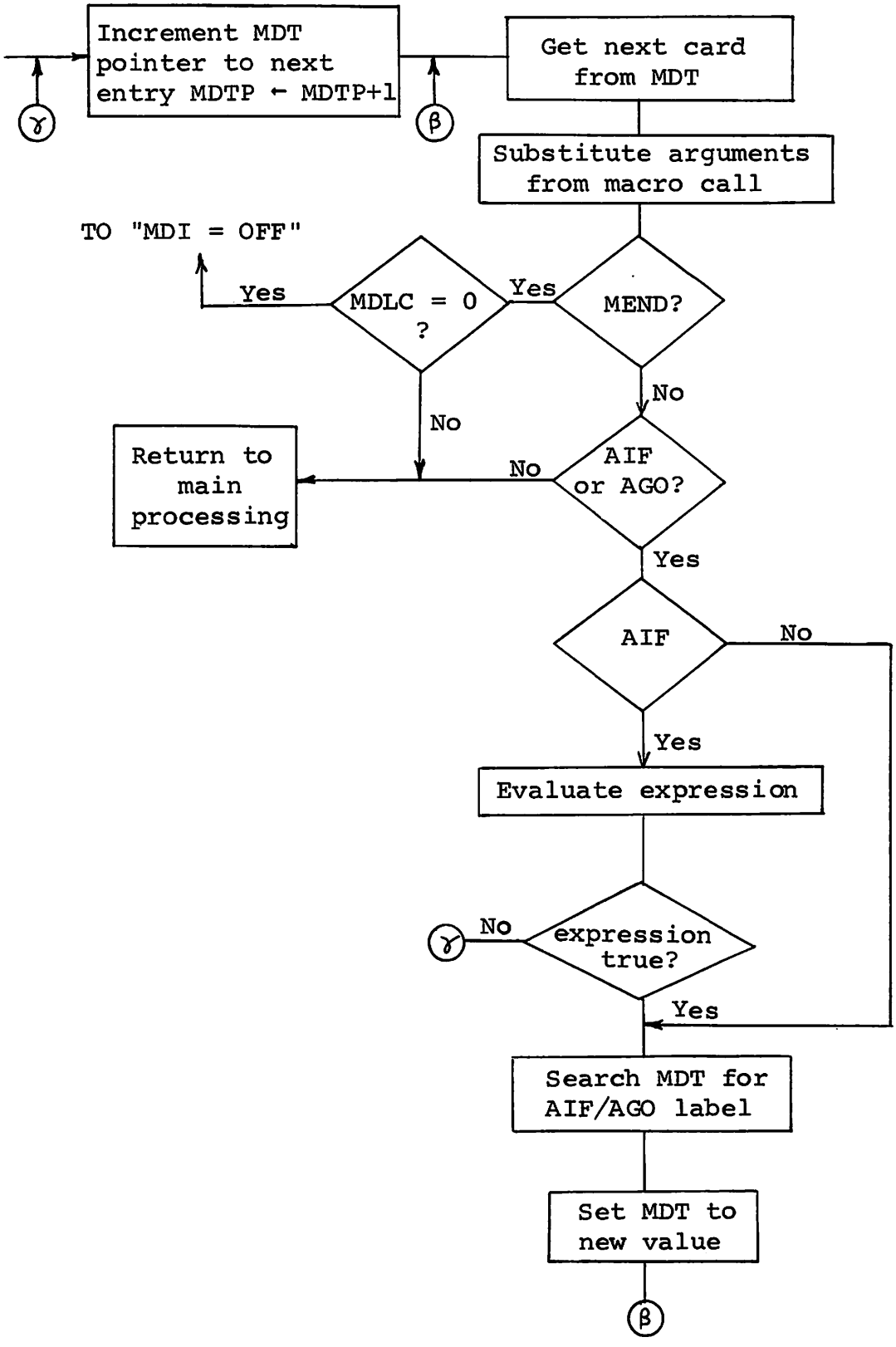
d. 1) An LCLA statement would result in an entry in the MST, setting the LCL/GBL falg to LCL, and assigning a new pointer to the symbol.

2) A GBLA statement will result in an entry with the LCL/GBL flag set to GBL, followed by a search to determine whether any identical symbol has the GBL flag set. If so, the value pointer of that symbol is copied. Otherwise, a new pointer is assigned.

10. i) Two pass model (change to Figure 6.2).



ii) One pass model (Figure 4.4).



iii) Assembler model modification is same as one-pass model except that MDTP is replaced by S(SP+1).

CHAPTER 5: LOADERS

1.
```
                EXTRN   SQRT
                  .
                  .
                  .
LABEL           AR      5,2
                  .
                  .
                  .
LOOP            B       LABEL
                  .
                  .
                  .
                END
```

a. A(LABEL) or A(LOOP)
b. A(LOOP-LABEL)
c. A(SQRT+LOOP)

2. a. Relocatable: Address is supplied by loader (linkage editor).

b. Absolute: The relative values for KOHN and LOOP in symbol table when subtracted produce absolute value to which is added absolute value of ONE from symbol table by assembler.

c. Relocatable: The relative address of PLACE (found in symbol table) is determined by the assembler; to this the loader adds the address of the beginning of the program.

d. Complex relocatable: The assembler places a minus relative value of LOOP for the address constant. The loader then subtracts from this constant the address of the beginning of the program, thereby setting the address constant to -(absolute value of LOOP). Then the loader adds the value of SIN to the address constant.

3. a. Binding is the process during which references between segments (procedures) are resolved, e.g., see answer to #2.

b. 1) On loading, values are placed in the transfer vectors.

2) At load time EXTRN values are supplied.

3) Addresses are supplied by the programmer or the assembler at assembly time.

4) When a reference is actually encountered at execution time.

5) At load time references can be resolved because overlay structure is known.

6) This is a binder capable of producing a relocatable load module from several segments. Binding occurs before loading, after assembling (compiling).

c. 1) Load time binding does not require the programmer to supply an absolute address, so the programmer's task is simplified and changing program locations does not necessitate re-assembly.

2) The scheme is more complex, requiring a larger loader and more time.

d. 1) Programs are not bound unless needed, and the name of a subroutine may be computed and supplied during execution rather than while programming.

2) This requires specialized hardware and a substantial amount of time overhead.

4. RLD - contains the information as to which locations must be either relocated or "fixed up" and how.
ESD - contains all information necessary for binding.
TXT - contains the object code of the segment.
END - signals end of deck to be loaded.

5. ESD
TXT
RLD
END

ESD updates the Global External Symbol Directory, allowing the
TXT cards to be loaded with any possible external references patched, after which
RLD can change any necessary locations and
END stops the process for this set of cards.

6. The ID number is a uniform symbol as opposed to the varying length names it represents. No entry in the RLD is made for locally defined symbols (LD) so there is no need for an ID.

7. a. The RLD would become unnecessary; all relocation could be handled by base registers.

   b. Pass 1 could not be simplified, as its function is to collect external references. Pass 2, however, would be relieved of the task of relocating the program. It would merely load the code as if it were absolute.

8. a.

```
PGA     length:   16
    PGA1      offset:  8
PGB     length:   20
    PGB1      offset:  0
    PGB2      offset:  8
    PGB3      offset: 16
PGC     length:   16
    PGC1      offset:  4
    PGC2      offset: 12
```

PGC must be displaced 4 bytes past the end of PGB for alignment on a double word boundary.

GEST

Symbol	Absolute Address
PGA	400
PGA1	408
PGB	416
PGB1	416
PGC	440
PGC1	444
PGC2	452

   b. PGB2 and PGB3 are local symbols.

c.

Location	Contents
400	400
404	420
408	8
412	12
416	440
420	416
424	420
428	0
432	450
436	XXXX
440	12
444	440
448	452
452	448
456	XXXX
460	XXXX

9. a. Pass 1 scans all ESD cards to create the Global External Table.

   b. Pass 2 loads, relocates, and inserts the values of external references in each segment.

   c. To implement a one-pass loader similar to the one described in the text, it would be necessary to load object decks in such a way that all external symbols are defined by being in a previously loaded program before being referenced. This is not always possible.

      Alternatively, a chaining procedure could be set up to "patch up" a previously loaded program. Since this requires returning to previously loaded material though, it is not really one-pass, but perhaps one-and-a-half-passes.

   d. Exactly the same as the two-pass DLL.

   e. Same as flowchart of loader (Figures 5.24 and 5.25).

   (Another approach to answering parts c, d, and e would be to restrict the assembly language, e.g., only allow external references to procedures that will be loaded first.)

10. a.

```
PGM     START   0
        EXTRN   SQRT
        EXTRN   TAN
        EXTRN   COS
        DS      5F
ENT1    DS      5F
ENT2    DS      10F
        DC      A(TAN+COS-ENT2)
        DC      A(COS)
        DC      A(ENT1-SQRT)
        END
```

b. No solution is not unique, small changes, such as

```
ENT1    DS      5D
```

instead of

```
ENT1    DS      5F
```

could be made without being reflected in the object deck.

11. a. The ABS card should be first in the deck taking the place of the START card. The location counter must be set to the correct value to create an absolute symbol table.

b. Both passes, as the location counter must be initialized at the beginning of each pass. At some point, as well, generation of relocatable code must suppressed.

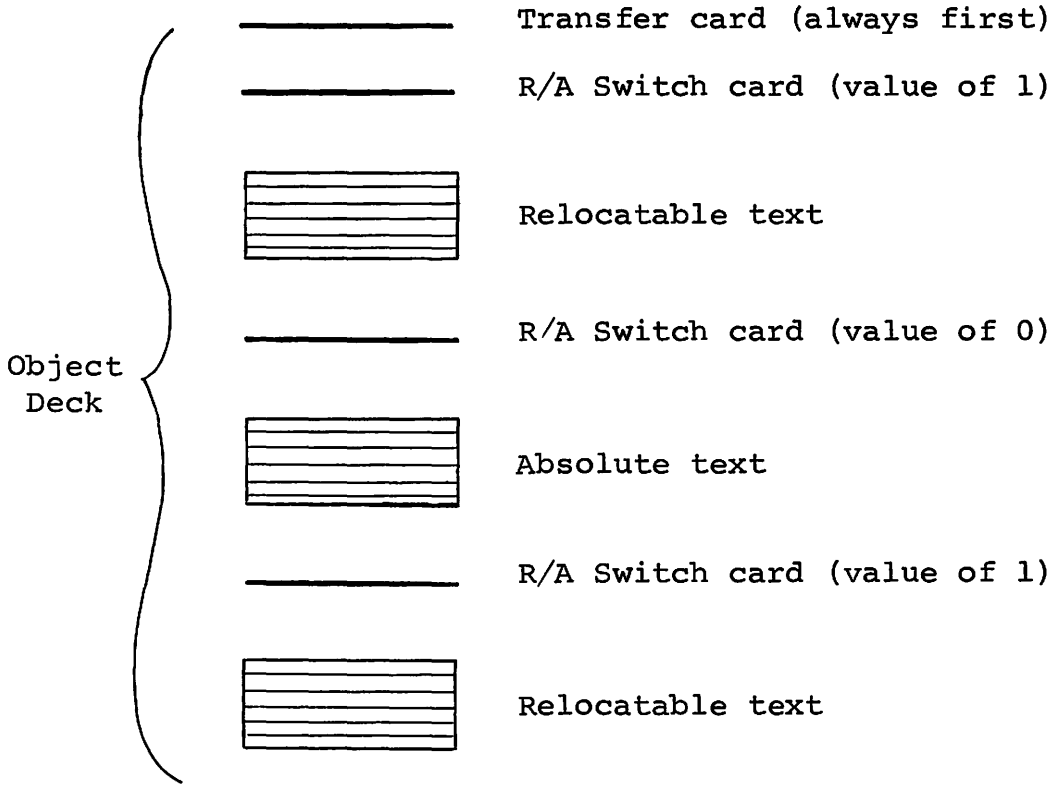
c. The data bases need not be changed in format.

d. The R/A flag can be retained to suppress output of relocatable code. At the beginning of the second pass, the ABS pseudo-op could be used to set all R/A flags in the symbol table to Absolute.

12. a. Add one card type to the absolute loader, the R/A switch card. A location internal to the loader will indicate whether text is to be treated as absolute or relocatable; this location will be set by the R/A switch cards:

Card Column	Contents
1	Card type = 2 (for R/A switch)
2	Count = 0
3	0 sets switch to Absolute 1 sets switch to Relocatable
4-72	empty
73-80	sequence number

b. All text after an R/A switch card will be treated as specified on that card until the next R/A switch card.



13. a. Object deck for segment SOLN

	Symbol	Type	ID	Relative Address	Length
ESD	SOLN	SD	01	0	50
	A	ER	02	-	-
	C2	ER	03	-	-

	Opcode	Relative Address	Address Constants	Actual Value of Address Constants
TXT	BALR	0	-	-
	SR	2	-	-
	SLL	4	-	-
	L	8	-	-
	BR	12	-	-
	DC	16	A(A+10)	10
	DC	20	-	-
	DC	24	A(C2)	0
	DC	28	-	-
	DC	40	-	-

	ID	Length	Flag(+/-)	Relative Address
RLD	02	4	+	16
	03	4	+	24

Object deck for segment GRADER

	Symbol	Type	ID	Relative Address	Length
ESD	GRADER	SD	01	0	60
	C2	LD	-	24	-

	Opcode	Relative Address	Address Constants	Actual Value of Address Constants
TXT	BALR	0	-	-
	SR	2	-	-
	L	4	-	-
	ST	8	-	-
	LA	12	-	-
	L	16	-	-
	BR	20	-	-
	DC	24	A(GRADER)	0
	DC	28	-	-

RLD

ID	Length	Flag(+/-)	Relative Address
01	4	+	24

b. 1) External references will be resolved where possible, and the relative address will be incremented on each program to create one series of TXT cards with origin at the beginning of the composite segment.

2) SD's will be downgraded to LD's.

3) The ID's must be modified to agree with the new ESD, and the relative address fields changed to agree with the composite TXT.

4) In the object deck for STUDENT, the ESP card for DELTA will still reference an external symbol.

5 & 6) Object deck for composite segment NEW.

ESD

Symbol	Type	ID	Relative Address	Length
NEW	SD	01	0	188
STUDENT	LD	-	0	-
A	LD	-	56	-
SOLN	LD	-	72	-
DELTA	ER	02	-	-
C2	LD	-	152	-
GRADER	LD	-	128	-

TXT

Opcode	Relative Address	Address Constants	Actual Value of Address Constants
DC	40	A(A+10)	66
DC	56	A(DELTA)	0
DC	64	A(-SOLN)	-72
DC	88	A(A+10)	66
DC	96	A(C2)	152
DC	152	A(GRADER)	128

	ID	Length	Flag(+/-)	Relative Address
RLD	01	4	+	40
	02	4	+	56
	01	4	-	64
	01	4	+	88
	01	4	+	96
	01	4	+	152

7) Two passes -- one to read ESD's, the second to modify TXT and write new ESD and RLD.

8) The input segments, a program library, a table of local values, and the composite ESD and RLD.

9) The output of a binder is a relocatable object deck; the output of a DDL is core image code in executable form.

10) This could be done, but the advantages are slight -- it doesn't happen <u>that</u> often -- and the cost in time and complexity is high.

14. a. The input to the de-linker will be assumed to be the composite segment and the names of the bound segments.

1) a) Restoring the old length field of an SD that became an LD is obviously easier if it is there. In addition, it may be the only way to get the original length, as each segment is bound to a double-word boundary.

b) Relative addresses of the LD's must be used rather than lengths of the original segments, because of double-word alignment.

c) Relocation also involves the relative address rather than the length, as does associating each composite RLD with a segment.

2) The original length could not be precisely determined; subtraction of relative addresses would give the length up to the next double word boundary.

3) The length is not necessary, as all segments are always loaded on double-word boundaries, and each de-linked segment will be identical to the original segment except for padding up to the first double word boundary, often the original end of the segment.

b. 1) FROM is not needed because it would contain the same symbol as the ones it describes (an old SD).

IN is needed in case of external references.

2) FROM cannot be used for an ER.

IN is needed.

c. Here we can eliminate the inputting of the bound segment names. If the FROM field is kept, it can be left blank for SD's that were downgraded to LD's, or filled in to indicate an old LD.

15. a. All binding can obviously be done in the binder, resulting in a simple relocatable load module. Or the relocating can be done in the binder as well, in which case the loader can be absolute.

b. See (a).

c. One. Nothing remains that required the second pass.

d. Relocatable, or absolute if the binder performs relocation. This scheme is sometimes referred to as a linkage editor. (See section 5.17 for variations of this scheme.

CHAPTER 6: PROGRAMMING LANGUAGES

1. a. Allocated at compile time. Freed only when program is no longer loaded.

   b. Allocated at execution time under program control. Freed under program control or when procedure is terminated (END or RETURN).

   c. Allocated on entering procedure. Freed when procedure is terminated.

   d. Same as (b).

2. Allocation and freeing of storage takes place in the same way. However, in controlled storage only the latest allocation is available, while in based storage any pointer-qualified allocation is available.

   Controlled storage may be conceptually considered a limited form of based storage.

3. A DO block is similar in some senses to a single statement. It may be the action-unit of a THEN or ELSE clause:

```
If A = B THEN DO;
              =
              =
              =
              =
              END;
         ELSE GO TO LABEL;
```

   In addition, a DO block may be iterative (DO I = 1 TO 20 BY 2;).

   A BEGIN block sets aside a block of code with the same effect upon scope of variables as a procedure block, but which is executed when encountered in the normal flow of control.

   A PROCEDURE block allows symbols to be defined within it irrespective of variables declared externally; while external variables may also be referenced. Execution skips over a procedure block encountered in the normal flow of control; a procedure is executed only when called.

   They are not interchangeable.

4. A DO group has no effect on the scope of a variable.* It begins when the DO statement is encountered and executes sequentially until the END statement (once or repetitively). Control then proceeds to the next statement unless the DO group has been used as the THEN unit of an IF statement, in which case the ELSE unit is skipped.

   A PROCEDURE, called or function, and BEGIN block, when located within another block will have no effect on the scope of variables unless an explicitly declared variable in the internal procedure has the same name as a variable in the external procedure, in which case the external declaration will be inaccessible within the internal block.

   A BEGIN block starts with the encountering of a BEGIN statement and continues until the complementary END statement.

   A called procedure is transferred to by a CALL statement. A function procedure is implicitly transferred to by a reference within an assignment statement. Execution of either may be terminated by an END or RETURN statement.

5. In addition to the control described in problem 6.4, the user can declare a variable EXTERNAL, making it available to other procedures, or INTERNAL, with no meaning outside his procedure.

   Programs may be written in sections by different programmers without having to worry about conflict of names.

6. 1) Transferred to an internal procedure simply by virtue of scope of declaration.

   2) As part of parameter list of PROCEDURE and CALL (or function reference) statements.

   3) Declared EXTERNAL in both procedures.

*An interesting point is the effect DO's have on "scope of labels, i.e., you cannot transfer to a label within an iterative DO group.

7.

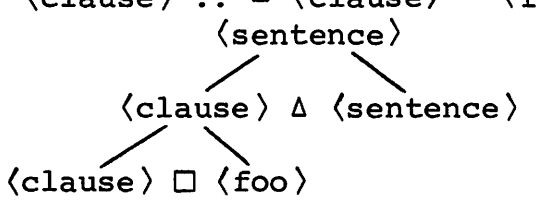
Statement Executed		Changed Values
(1)		
(2)		X = -2
		Y = -2
		W = 0
		Z = 0
(8)		M allocated
(9)		M = 4
(10)		Z = -4
(11)		
(12)		
(13)		W^1 = 2
(14)	(3)	
	(4)	
	(5)	X^1 = -4
	(6)	Z = 14
(15)		
(16)		
(19)		
(20)		X = 0
(21)		
(22)		Z = 0
		Y = 0
		W = 0
(23)		

8.

(1)		
(2)		
(3)		Z = 12
(4)		
(5)		
(6)		X = 5
(7)		ZED = 17
(8)		
(9)		
(10)		X = 1
(11)		
(12)		
(25)	(37)	
	(38)	Z = 1
	(39)	Z = -39
	(40)	
(41)	(35)	
	(36)	X = 5
	(37)	
	(38)	Z = 25
	(39)	Z = -15
	(40)	
(42)	(26)	
	(27)	
	(28)	
	(29)	Z^1 allocated
	(30)	X = 4
	(31)	Z^1 = 16
	(32)	X = 20
	(33)	Z^1 freed
(43)	(44)	
	(45)	
	(46)	STR = 60
	(47)	Z = 80
(49)		

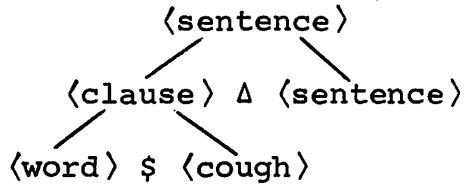
1. a. 1) ⟨sentence⟩ :: = ⟨clause⟩ Δ ⟨sentence⟩ right
2) ⟨clause⟩ :: = ⟨clause⟩ □ ⟨foo⟩ left
4) ⟨word⟩ :: = ⟨word⟩ $ ⟨char⟩ left
6) ⟨cough⟩ :: =⟨cough⟩ Δ ⟨char⟩ left

b.‡ i) ⟨sentence⟩ :: = ⟨clause⟩ Δ sentence
⟨clause⟩ :: = ⟨clause⟩ ⟨foo⟩



□ has precedence over Δ

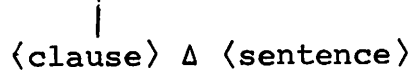
ii) ⟨sentence⟩ :: = ⟨clause⟩ Δ ⟨sentence⟩
⟨clause⟩ :: = ⟨word⟩ $ ⟨cough⟩



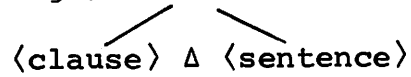
$ has precedence over Δ

‡

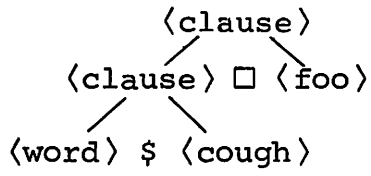
NOTE: In answers involving a tree, for clarity we have drawn the single substition, e.g., ⟨sentence⟩



with two branches, e.g., sentence

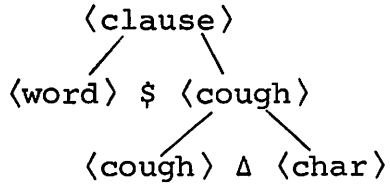


iii) ⟨clause⟩ :: = ⟨clause⟩ □ ⟨foo⟩
⟨clause⟩ :: = ⟨word⟩ $ ⟨cough⟩



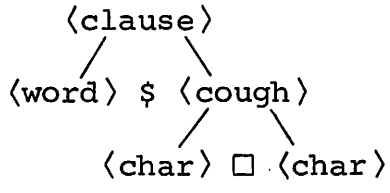
$ has precedence over □

iv) ⟨clause⟩ :: = ⟨word⟩ $ ⟨cough⟩
⟨cough⟩ :: = ⟨cough⟩ Δ ⟨char⟩



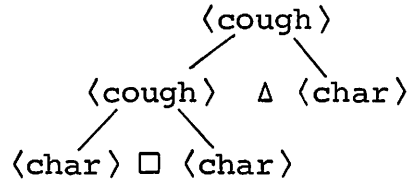
Δ has precedence over $

v) ⟨clause⟩ :: = ⟨word⟩ $ ⟨cough⟩
⟨cough⟩ :: = ⟨char⟩ □ ⟨char⟩



□ has precedence over $

vi) ⟨cough⟩ :: = ⟨cough⟩ Δ ⟨char⟩
⟨cough⟩ :: = ⟨char⟩ □ ⟨char⟩



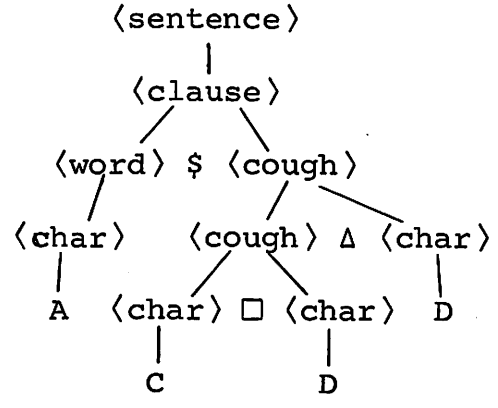
□ has precedence over Δ

c) Summary of part (b)

i) □ > Δ
ii) $ > Δ
iii) $ > □
iv) Δ > $
v) □ > $
vi) □ > Δ

It would appear that (ii) contradicts (iv) and (iii) contradicts (v). The relationship implied by (ii), for instance, applies only to a string generated by applying the rewrite rules shown. The relationship in (iv), which is the reverse, generated by applying different rewrite rules. The resulting strings will be different in each case, and there will be no ambiguity or inconsistency.

d)



A $ C □ D Δ D

Note that there is no way to generate a tree with precedence of □ and Δ or Δ and $ reversed.

2. 1) To specify a language
2) To recognize strings of a language
3) In investigating the complexity of a language
4) To show the possible equivalence of programs
5) To compare languages

See section 7.1 for detailed explanation.

3. a. 2) A language is a set of strings of symbols, with structure and (usually) meaning.

   1) A grammar is a set of rules that may be used to generate a language.

   3) A machine or automation is a device that recognizes and accepts strings of a given language.

   b. A BNF specification is a grammar. A canonic system is also a grammar although potentially more descriptive.

   c. A language capable of being described by BNF can always be described by a canonic system. The reverse is not necessarily true; a canonic system can describe (context-sensitive) language features that a BNF cannot. Specifically, BNF is equivalent in power to a restrictive class of canonic systems.

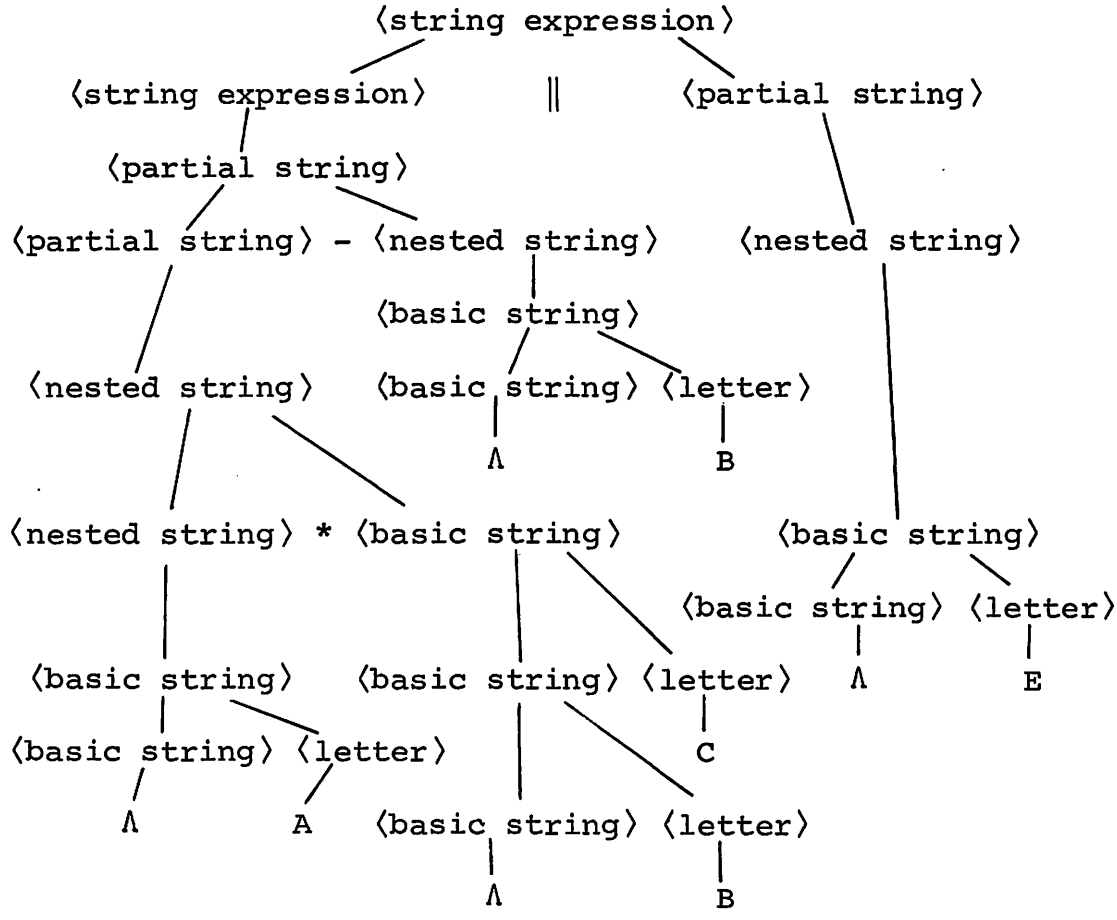
4. A generative grammar may have to generate any number of strings before it matches the string in question or exhausts the entire collection of equal-length strings to prove the string is not legal. This takes time.

5. a. BNF is incapable of describing the context-sensitive features of a language.

   b. Canonic systems can specify sets and relationships sufficient to describe context-sensitive features (such as the in relationship for labels; see 7.6.1).

6. a. Of course, there are a number of examples. Two are shown below. Both use each rule at least once (except for the letter rules; to use each letter in an exercise would be absurd).

i)

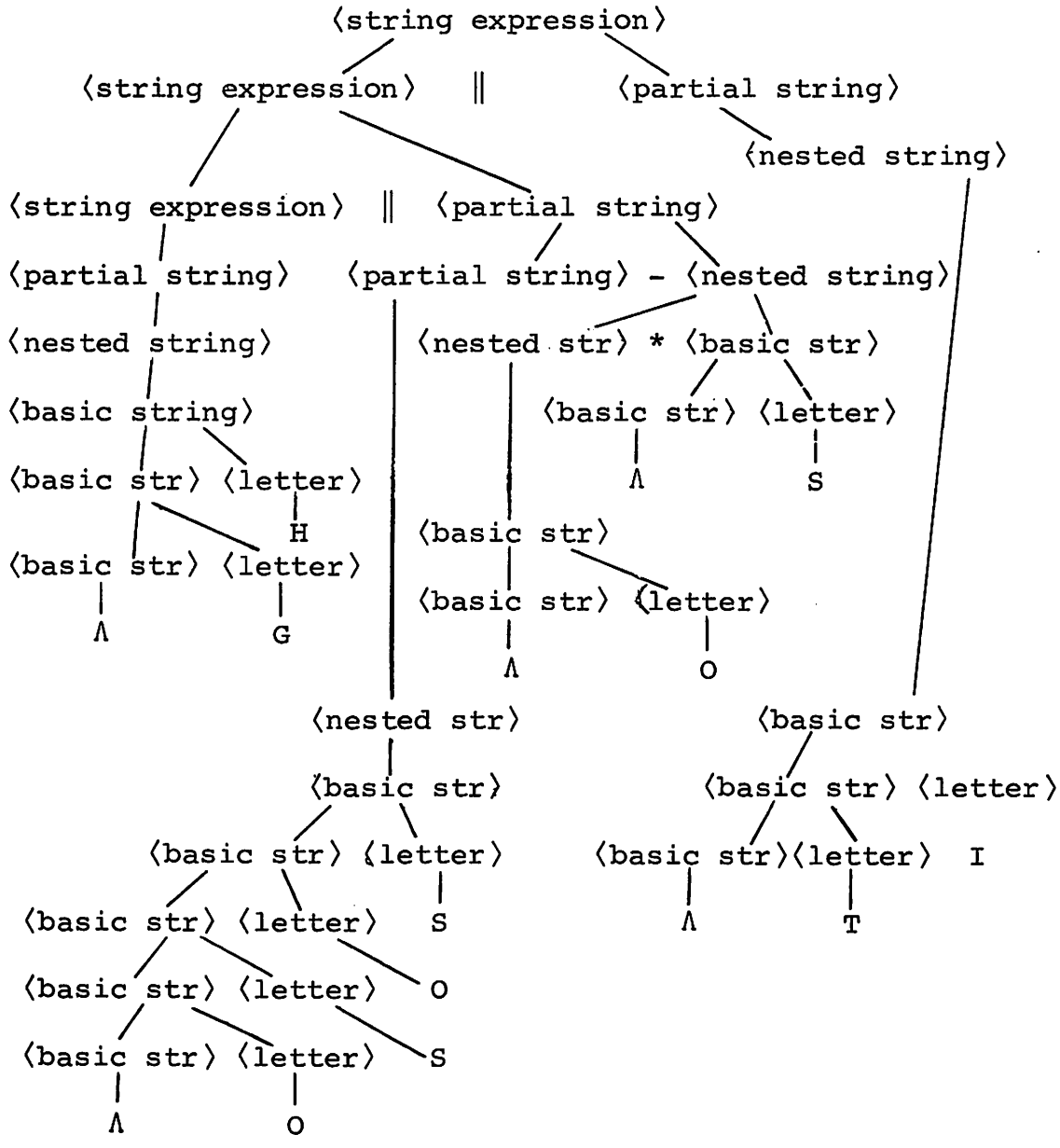


A * BC - B || E (= BCABC - B || E

= CAC || E

= CACE)

ii)



GH || OSOS - O * S || TI

GH || OSOS - O * S || TI

(= GH || OSOS - SOS || TI

= GH || O || TI)

= GHOTI

which is pronounced "fish".

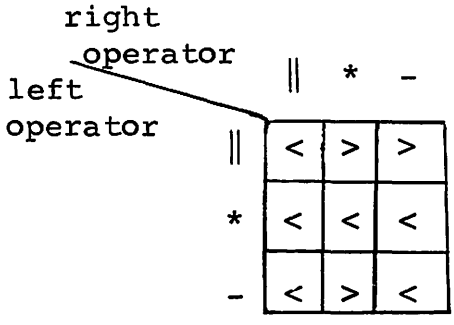
(GH as in LAUGH

O as in WOMEN

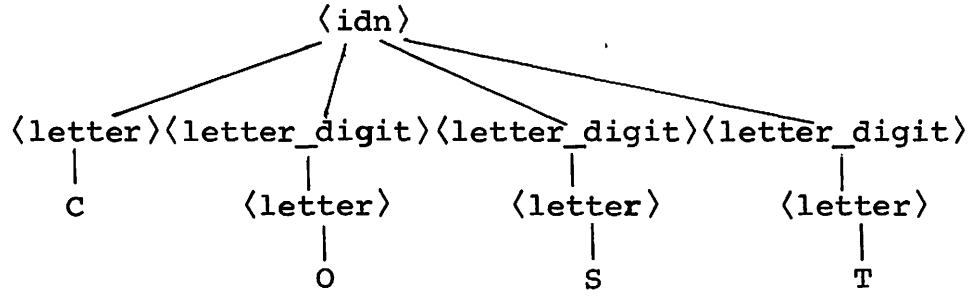
TI as in COMMOTION)

b) * > - > ||

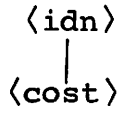
c)



7. a. All derivations from ⟨idn⟩ to a symbol are to this form:

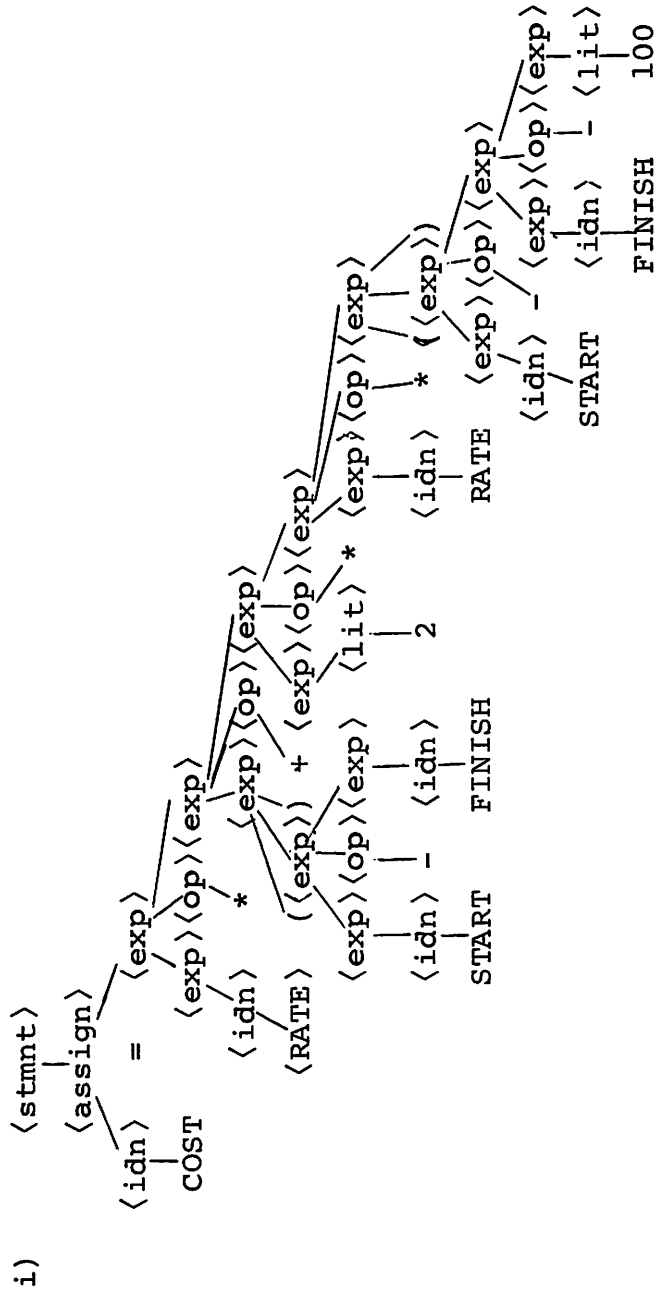


They will be abbreviated:

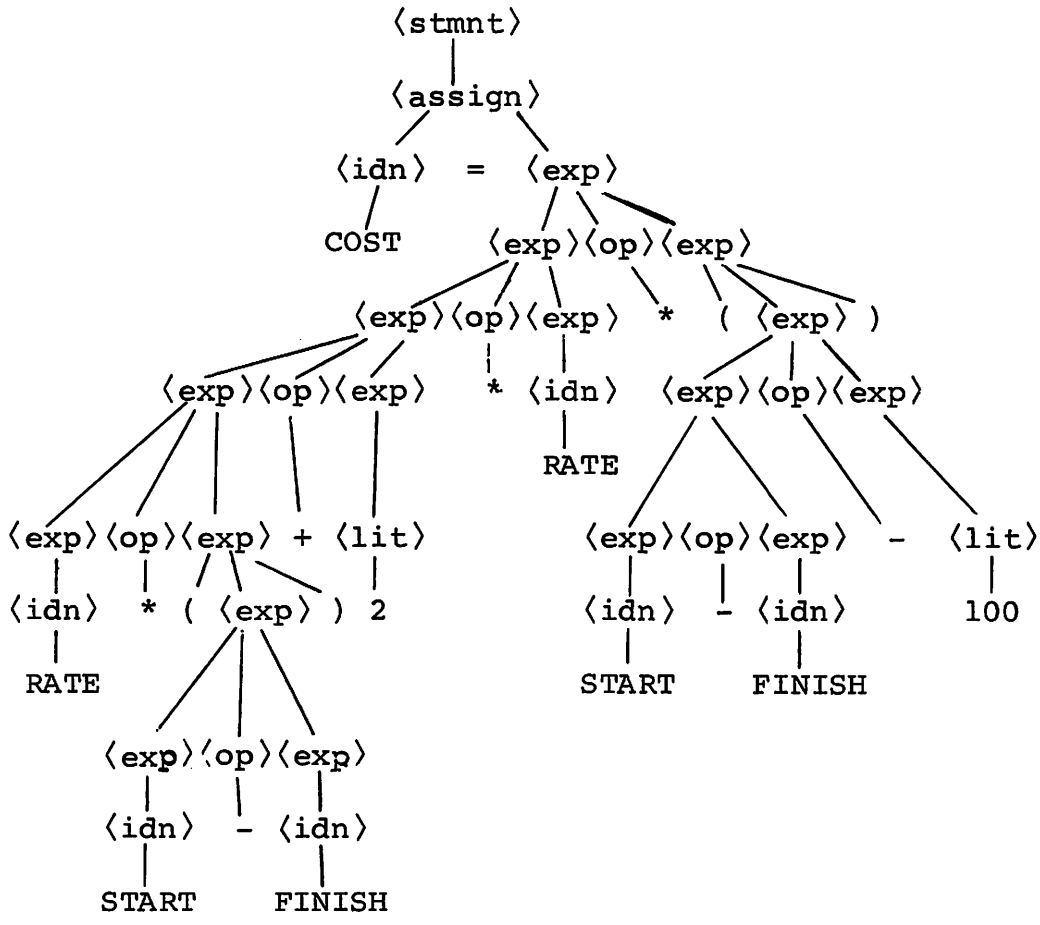


There are many different parse trees for this statement. Two of them are as follows:

i)


⟨stmnt⟩
⟨assign⟩
⟨idn⟩ = ⟨exp⟩
COST ⟨exp⟩⟨op⟩⟨exp⟩
⟨idn⟩ * ⟨exp⟩⟨op⟩⟨exp⟩
⟨RATE⟩ (⟨exp⟩) + ⟨exp⟩⟨op⟩⟨exp⟩
⟨exp⟩⟨op⟩⟨exp⟩ ⟨lit⟩ * ⟨exp⟩⟨op⟩⟨exp⟩
⟨idn⟩ - ⟨idn⟩ 2 ⟨idn⟩ * (⟨exp⟩)
START FINISH RATE ⟨exp⟩⟨op⟩⟨exp⟩
⟨idn⟩ - ⟨exp⟩⟨op⟩⟨exp⟩
START ⟨idn⟩ - ⟨lit⟩
FINISH 100

ii)



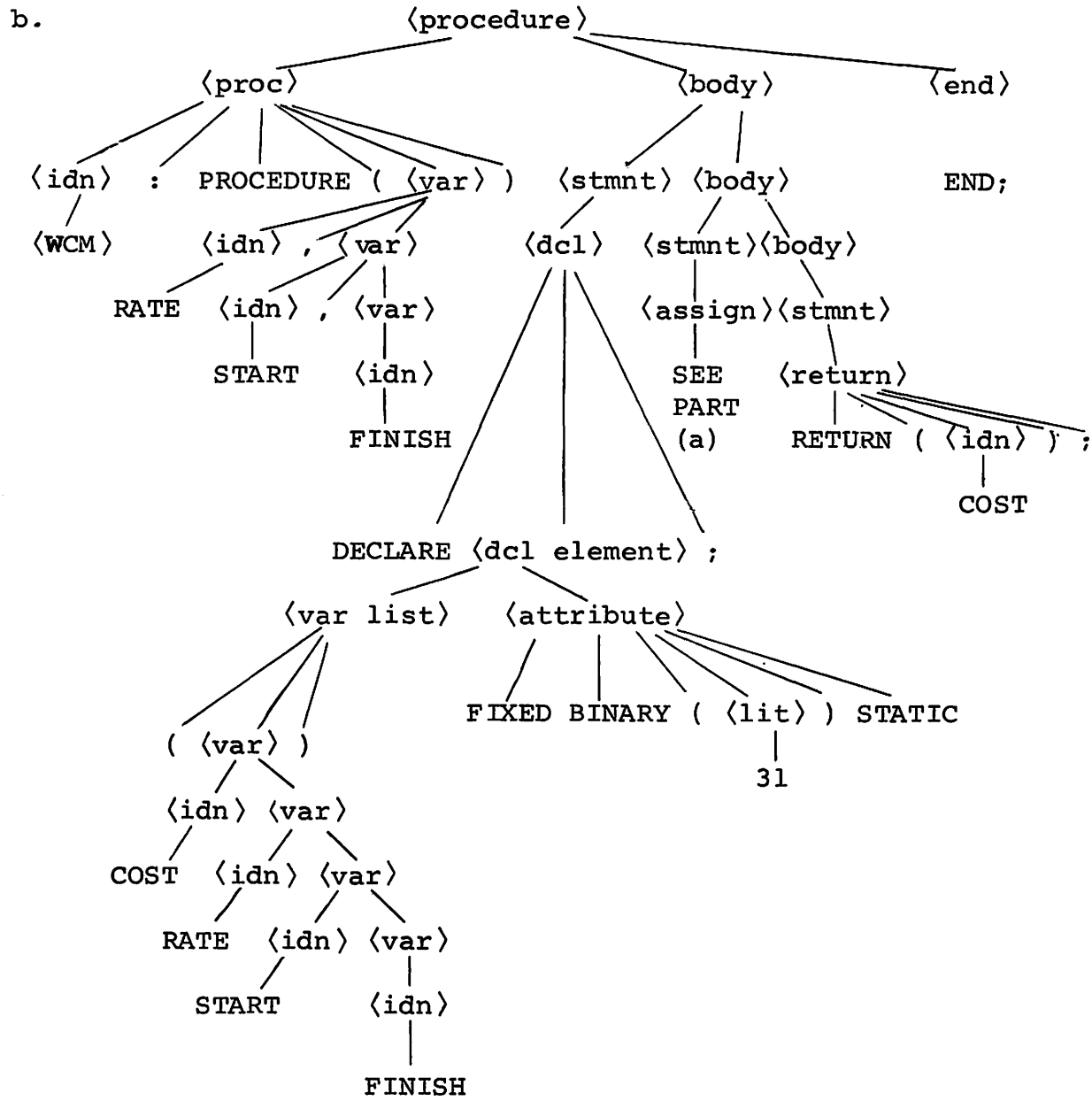
NOTE that (i) is equivalent to:

```
COST = RATE *((START-FINISH)+(2 * (RATE *(START-(FINISH-100)))))
```

while (ii) is equivalent to

```
COST = ((((RATE *(START-FINISH))+2) * RATE) *((START-FINISH)-100)
```

b.



c. There is no established precedence, as shown by the two legitimate derivations in part (a). Only complete parentization (which is legal) can specify the order of computation unambiguously.

d. Eliminate:

⟨exp⟩ :: = ⟨lit⟩ | ⟨idn⟩ | ⟨exp⟩ ⟨op⟩ ⟨exp⟩ | (⟨exp⟩)
⟨op⟩ :: = + | - | * | / | * * | =

Add:

⟨sym⟩ :: = ⟨lit⟩ | ⟨idn⟩ | (⟨exp⟩)
⟨maze⟩ :: = ⟨sym⟩ * * ⟨maze⟩ | ⟨sym⟩
⟨term⟩ :: = ⟨term⟩ * ⟨maze⟩ |
⟨term⟩ / ⟨maze⟩ | ⟨maze⟩
⟨exp⟩ :: = ⟨exp⟩ + ⟨term⟩ |
⟨exp⟩ - ⟨term⟩ | ⟨term⟩

8. a. Canons 6 and 8 establish * as having greater precedence than +.

b. Change:

5) q res ⊢ q term
6) t term ; q res ⊢ t * q term

Add:

10) p primary ⊢ p res
11) p primary ; s res ⊢ p $ s res

9. a. ⟨letter⟩ :: = P | Q | R | S
⟨id⟩ :: = ⟨letter⟩ | ⟨id⟩ ⟨letter⟩
⟨primary⟩ :: = ⟨id⟩ | (⟨exp⟩)
⟨term⟩ :: = ⟨primary⟩ | ⟨term⟩ * ⟨primary⟩
⟨exp⟩ :: = ⟨term⟩ | ⟨exp⟩ + ⟨term⟩

b. No BNF equivalent
Canons not translatable: 5,6,7,8,9

QUESTIONS 7.10, 7.11, and 7.12 are left to the student as each answer will be different.

1. See Figure 8.13.

2. (translator, interpreter)
A translator inputs a program in one language and outputs a program in another; an interpreter inputs a program in a language and performs the intent of the program without producing object code (e.g., the 370 is an interpreter for machine code. Such languages as BASIC and APL are usually implemented using an interpreter.

   The syntax phase of a compiler may be thought of as an interpreter, i.e., the syntax phase interprets reductions. (Note: the reductions could be compiled if one wrote a translator for reductions.

3. (pass, phase)
A pass is a physical traversal of the input deck or a data base; a phase is a logical segment of a compiler, performing one conconceptual function (e.g., the optimization phase may consist of many passes, where the two phases, syntax and interpretation, may be implemented as a single pass.

   (syntax analysis, semantic interpretation)
Syntactic analysis is based on structure alone; semantic analysis denotes analysis on the basis of meaning as well.

   (token, uniform symbol)
A token is a lexical unit, part of the input. A uniform symbol is a compiler-generated unit to represent, in fixed format, the variable format tokens.

4. Advantages: It is faster, since the compiler has already performed many of the tasks of an assembler (e.g., created a symbol table, literal table). These functions would not have to be repeated by the assembler.

   Disadvantages: It requires a larger, more complex compiler and may take more core at one time.

5. A uniform symbol is much easier to handle than a variable format datum. It can be avoided, but then program complexity must be increased to handle the different lengths of symbols and the different lengths of data that may be passed on to each phase.

6. The idea is that each phase performs some unique function that can be understood more easily as a separate unit. A compiler, for instance, could be considered as a one-phase program, the compile phase. But this doesn't say anything about its construction. Neither does the idea of a "matrix-generation" phase say enough to someone trying to understand (or design) the compiler.

7. The lexical, syntax, interpretation, and optimization phases are by and large independent of a machine (i.e., a compiler running on an IBM 360 could produce code for many other computer with no modification to these phases). The data bases of these phases are machine independent. A literalist would point out that the last three phases also use the identifier table and literal table, which may be machine independent. But this evades the spirit of the question. The idea is that only the code production and the assembler phase data bases are machine independent.

8. No. This is a tradeoff between compilation time and execution time. A program that will be run only once for debugging, for instance, is not worth the investment of compilation time. Sometimes a parameter is available to allow the user to specify the degree of optimization he desires.

9. Productions generate strings. Reductions parse (break up) strings, e.g., BNF productions can be used to generate a PL/I program. Reductions of a compiler may be used to parse a PL/I program.

10.

	Created	Permanent	Lang Dep	Mach Dep	Lang Indep	Mach† Indep
Terminal Table		X	X			X
Identifier Table	X		X			X
Literal Table	X		X			X
Reductions		X	X			X
Uniform Symbols	X		X			X
Matrix	X		X			X
Matrix Operators		X	X			X
Code Productions		X	X	X		
Action Routines		X	X			X

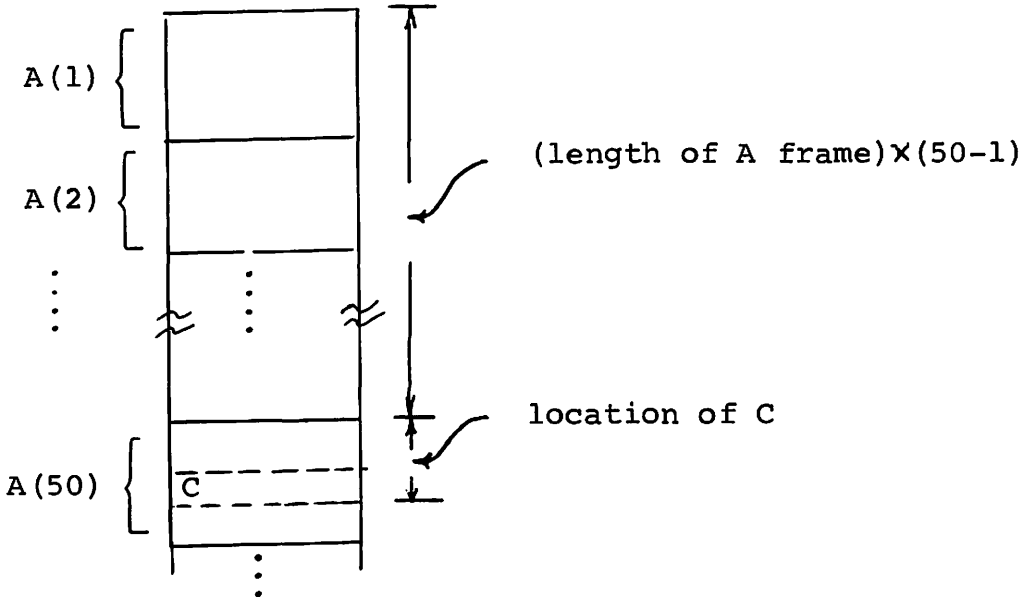
†In reality, many of these data bases may depend slightly on the machine or their formats and contents are partially influenced by the machine.

11. Change the code productions and the assembler phase to produce 645 code.

12. The interpretation phase creates matrix entries.
The machine-independent optimization phase may modify the machine (e.g., delete entries).
The code generation phase uses the matrix without modifying it.
The storage assignment phase adds entries pertaining to the storage, e.g., | static | 10 | |

13. The location, precision, and data type of X.

To formulate the access of A.B.C(50), we need the data type, precision, and location of C. The location may be in the location of C within one of the A's. We also need the length of one of the A's so as to multiply that length by (50 -1). Thus the address of C = (location of C + (length of one A frame) x (50 - 1). This assures a storage allocation as follows.



14. ///*/ should be //*/

    ⟨idn⟩ /"/*/2 should be ⟨idn⟩ //"*/2

15. Permanent Tables: Terminal Table
    Reductions
    Code productions

    Created Tables: Uniform Symbol Table
    Identifier Table
    Literal Table
    Matrix

16. Canonic systems and BNF are generative specifications of the language. BNF cannot describe the context-sensitive features of languages. Reductions parse strings.

17. A PROCEDURE statement may generate as simple a matrix entry as

    procbgn ⟨idn⟩

    Attributes such as RECURSIVE might cause a more elaborate matrix entry, in order to set up code in the prolog to handle recursion.

    DCL does not directly result in any entry in the matrix. When processing the DCL statement, the interpretation phase simply places the appropriate information into the identifier table. Later, however, the storage assignment phase must assure that the storage will be assigned, and if the initial attribute was used, the storage must be initialized accordingly. To accomplish this, when scanning through the identifier table, the storage phase will generate appropriate matrix entries for code generation to have the storage assigned and initialized.

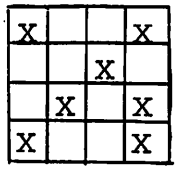
18. The code generation patterns for (+) serve as a model as adequately as one that could be given here for (-).

19. Register use optimization
    Elimination of redundant load and store instructions
    Use of more efficient instructions, i.e., shorter (RR instead of RX), faster (shift left 1 instead of multiply by 2)

    All these types of code optimization are machine dependent and therefore are performed in the code generation phase. The disadvantage of performing them earlier is that the early phases would then become machine dependent.

20. In the optimization phase, because it is a totally machine independent optimization that can be most efficiently performed on the matrix.

21. a.



b. Common subexpression elimination, machine independent optimization phase eliminating matrix entries.

Compile time compute, machine independent optimization phase substituting a literal for several matrix entries.

Boolean expression evaluation, machine independent optimization phase substituting matrix entries to circumvent unnecessary operations.

Code expansion, code generation phase, substituting a macro for a calling sequence.

22. There are two cases here -- one involving the unary operator*(-) and one the binary operator (-).

Case 1: - (A-B) + X
B-A + X

-	B	A
+	M1	X

-	A	B
-	0	M1
+	M2	X

* See note on following page

Case 2: X - (A - B)
X + B - A

+	X	B
-	M1	A

-	A	B
-	X	M1

*(There is no mention of the unary operator in the text. There might be a special sign to indicate the unary minus to take advantage of special machine instructions, such as load complement register [LCR] OR 0 might simply be assumed as the other operand, as was done here. The latter is more generally applicable.)

In Case 1 the problem is one of recognition. A special check for pairs of matrix entries, one of which involves the unary minus, would have to be made. After that, the procedure would be that outlined in the text.

Case 2 is obviously much more difficult due to the order of evaluation. The optimization here is difficult to check for.

23. a, b, and c: There are several ways to implement a facility to handle several arithmetic types. One way is to leave the lexical, syntax, and interpretation phases essentially the same as described in the text. (The action routine to enter the data type of variables into the identifier table would enter more than FIX BIN). The code generation phases would use the entries in the identifier table to generate the proper code. To find the appropriate entries in the identifier table, code generation would use the uniform symbol in the operands to index the identifier table.

Another approach would be to change the interpretation phase to recognize the data types of variables used in an arithmetic expression and enter a different operator in the matrix, depending on the data type defined (e.g., | + FLOAT | A | B | for A + B where A and B are floating point numbers). Then the code generation phases would have appropriate macro definitions defined.

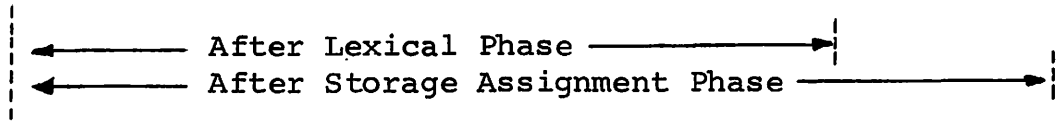
Assuming that mixed mode expressions are not allowed, another approach would be to add an additional column in the matrix to specify the type of data involved in each operation. This could be done by the action routines when making the matrix entries by checking the data type in the identifier/literal table. This would not involve the reductions; they operate purely on the strings, and there is no way of determining type from the strings.

d. Using any of the approaches above, the code generation would have to change, as shown in the example. However, the similarity of FIXED BINARY expansions to those for FLOAT BINARY suggests that one macro could be used for both, with modification to the op codes made for each. FIXED DECIMAL obviously is a totally separate expansion.

e. For mixed mode expressions the matrix could contain a "data type" column for each operand, and conversion would be decided at the code generation phase. Another approach would be to have more complicated macro definitions.

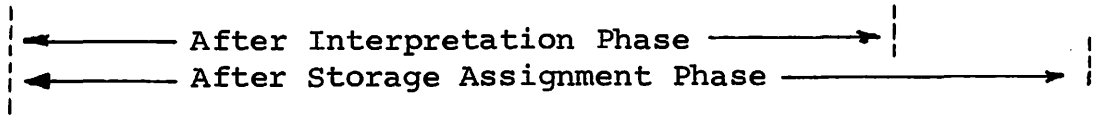
24. 1) a. LITERAL TABLE

Literal	Base	Scale	Precision	Address
10	BINARY	FIXED	15	0



IDENTIFIER TABLE

Name	Base	Scale	Precision	Storage Class	Address
BEGIN	Label, procedure name				
A	BIN	FIXED	15	STATIC	0
B	BIN	FIXED	15	STATIC	2



b.

```
;
END
PROCEDURE
```

c. An asterisk indicates that a match with the stack was made; the number denotes the reduction of Figure 8.17.

1*	18	8
2*	19*	9
4*	7	10*
4	8	
5*	9*	
7*	12*	
14	7	
15*	8	
15	9*	
16*	12*	
14*	7	

d. Matrix

1)	procbhn	BEGIN	
2)	arg	A	
3)	arg	B	
4)	=	A	10
5)	=	B	A
6)	procend		
	storage assignment info.		

e. Assume these simplified linkage conventions:

i) Return address in calling program is in register 15
ii) Pointer to location of value returned will be placed in register 14

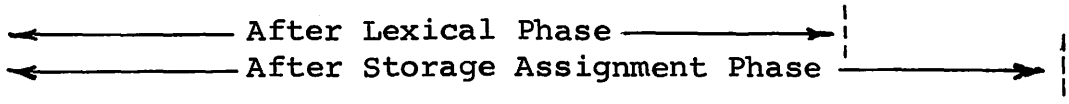
```
                                                From Matrix Entry
BEGIN    START   0
         STM     15,13,SAVE  ]
         BALR    15,0        } 1
         USING   *,15        ]
         ENTRY   A           } 2
         ENTRY   B           } 3
         LH      1,=H'10'    ]
         STH     1,A         } 4
         LH      1,A         ]
         STH     1,B         } 5
         LM      15,13,SAVE  ]
         BR      15          } 6
A        DS      H           ]
B        DS      H           } from storage
SAVE     DS      15F         ] assignment
         LTORG               ]
         END                 } 6
```

f. Either: 1) Have op_prec call error if it detects an error

or 2) Modify op_prec to place ⟨result⟩⟨error flag⟩ on top of stack. Then add reduction 12.5, ⟨result⟩⟨error flag⟩/error//7

2) a. LITERAL TABLE

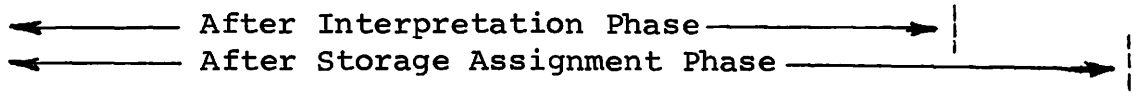
Literal	Base	Scale	Precision	Address
10	BINARY	FIXED	15	0



IDENTIFIER TABLE

Name	Base	Scale	Precision	Storage Class	Address
MIT	Label, procedure name				
JOHN	BINARY	FIXED	31	STATIC	0
STU	BINARY	FIXED	31	STATIC	4
BILL	BINARY	FIXED	15	STATIC	8

After Lexical Phase



b.

```
;
END
PROCEDURE
```

c. Asterisk indicates match to top of stack was made:

1*	18*	7
2*	14*	8*
4*	18	5*
4	19*	7
4	7	8
5*	8	9
7*	9*	10*
14	12*	
15*	7	
15	8	
16*	9*	
14*	12*	

d. Matrix

1)	procbgn	MIT	
2)	arg	JOHN	
3)	arg	STU	
4)	arg	BILL	
5)	*	STU	BILL
6)	=	JOHN	M5
7)	*	BILL	10
8)	-	JOHN	M7
9)	=	STU	M8
10)	rtn		
11)	arg	STU	
12)	procend		
	storage assignment info.		

e.

			From Matrix Entry
MIT	START	0	
	STM	15,13,SAVE	1
	BALR	15,0	
	USING	*,15	
	ENTRY	JOHN	2
	ENTRY	STU	3
	ENTRY	BILL	4
	LH	1,BILL	
	SR	0,0	5
	M	0,STU	
	ST	1,M5	
	L	1,M5	6
	ST	1,JOHN	
	LH	1,=H'10'	
	SR	0,0	7
	MH	0,BILL	
	ST	1,M7	
	L	1,JOHN	
	S	1,M7	8
	ST	1,M8	
	L	1,M8	9
	ST	1,STU	
	LA	14,STU	10,11
	LM	15,13,SAVE	12
	BR	15	
JOHN	DS	F	
STU	DS	F	
BILL	DS	H	from
M5	DS	F	storage
M7	DS	F	assignment
M8	DS	F	
SAVE	DS	15F	
	LTORG		12
	END		

NOTE: An efficient storage assignment phase would have assigned the same location to both M5 and M7 since they are serially reusable.

25. a. Lexical and syntactic phases. The lexical phase can no longer use break characters from the terminal table to perform final lexical analysis, i.e., does lexical phase replace DO17I by a uniform symbol pointing to the IDN table or replace the DO by a uniform symbol pointing to the terminal table?

    b. There would have to be more interaction between phases. The lexical phase would have to be equipped to parse strings in different ways. In the example given, for instance, the first parse would be

    DO17I/ = /1/,/10/,/1

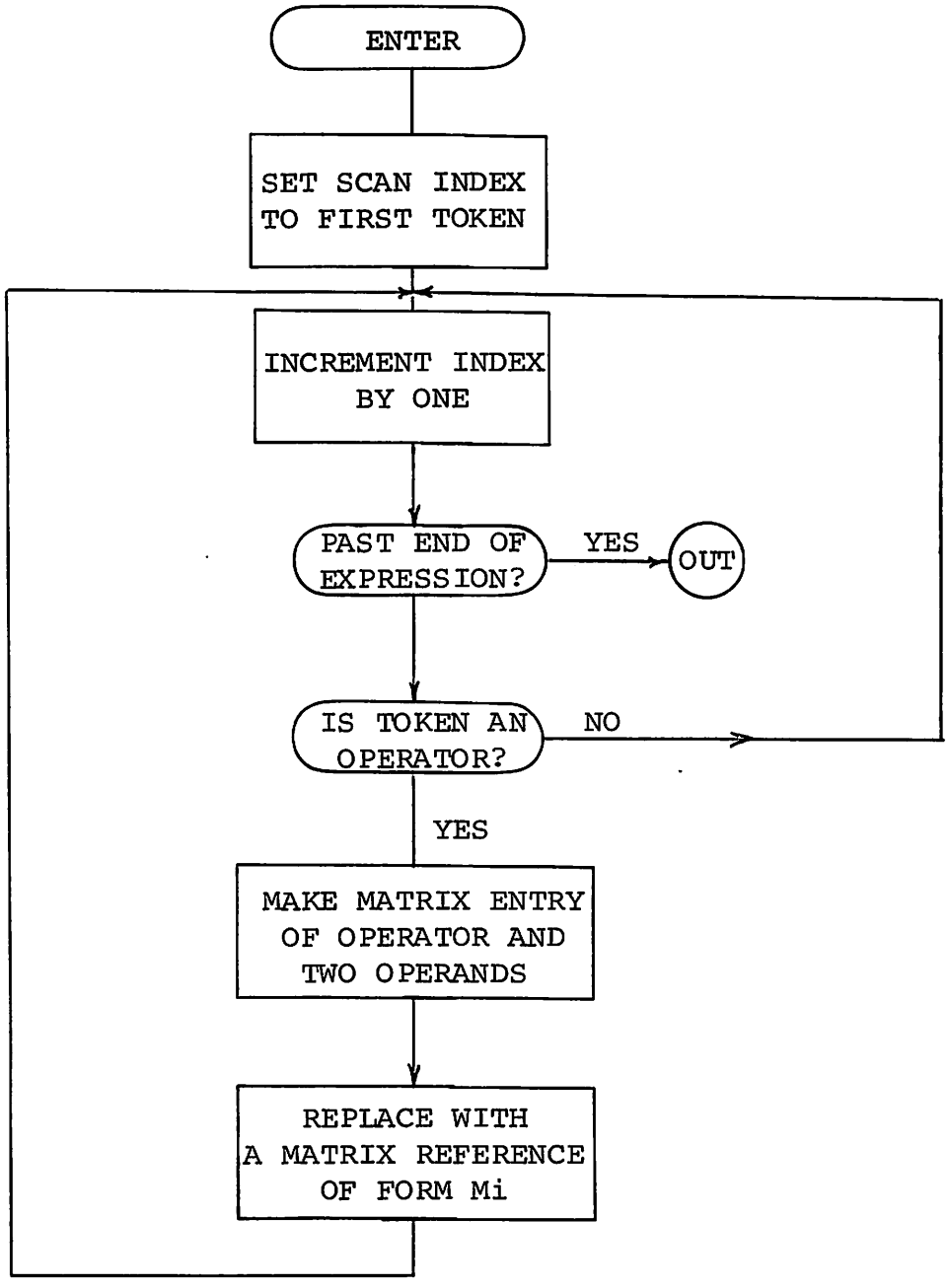
    The syntactic phase, on encountering an incorrect structure would return to the lexical phase rather than immediately taking error action. The lexical phase would then parse under different rules to produce

    DO/17/I/ = /1/,/10/,/1

    There may be times when an incorrect (i.e., not what the programmer intended) parsing is accepted as correct.

26. The problem of forward references (branches, storage location) would have to be worked out, probably with a loss of core usage efficiency. This resolution of forward references could be done using a transfer vector approach as was used in the BSS loader of Chapter 5. Only very limited optimization techniques would be possible to implement. Further restrictions would probably have to be placed on the language, e.g., DCL statements should appear before variables declared are used. A one-pass scheme would not have "logical modularity", e.g., the machine dependent parts would be interwoven with machine dependent parts.

27. a. No parenthesesization



b. ⟨idn⟩ = //***/

1 ⟨any⟩⟨op⟩⟨idn⟩ /mat_ent/*/

⟨any⟩ ; ///2

//*/1

2 ⟨idn⟩ = ⟨any⟩ ; /assign/"/

mat_ent creates an entry:

+ ⟨any⟩ ⟨idn⟩
Or - ⟨any⟩ ⟨idn⟩

and replaces ⟨any⟩⟨op⟩⟨idn⟩ in the stack with a matrix reference M_i

assign creates a matrix entry

= ⟨ian⟩⟨any⟩

c.
```
PARSE:  PROCEDURE (EXPRESSION);
DECLARE  1  MATRIX (100) EXTERNAL,
            2 OPCODE CHAR(1),
            2 OP1  CHAR(8),
            2 OP2  CHAR(8),
         MAT_INDX FIXED BIN EXTERNAL,
         SCAN FIXED BIN INITIAL (1),
         1 EXPRESSION BASED (P),
           2  SIZE FIXED BIN,
           2  TOKEN (J REFER (SIZE)) CHAR (8);
LOOP:  SCAN = SCAN + 1;
     IF SCAN > SIZE THEN RETURN; ELSE;
     IF TOKEN (SCAN) = '+' | TOKEN(SCAN) = '-'
        THEN DO:
             OPCODE(MAT_INDX) = TOKEN(SCAN);
             OP1 (MAT_INDX) = TOKEN(SCAN-1);
             OP2 (MAT-INDX) = TOKEN(SCAN+1);
             TOKEN (SCAN+1) = 'M' || MAT_INDX;
             MAT_INDX = MAT_INDX + 1;
             END;
        ELSE;
     GO TO LOOP;
END PARSE;
```

32. Data type, including length, precision, etc.; size, if an array; connectivity, if part of a structure.

    The executable program must know what to have allocated when allocation is requested. Note that this also applies to based variables.

33. No. This will handle all the variables, but for program control (like renumbering the number of levels of recursion), a stack frame is needed (check the PL/I RECURSIVE attribute).

34. a. The interpretation phase could check, upon making the matrix entry, against the identifier table. The simplest corrective measure would be to reduce the subscript to the maximum declared value.

    b. The lexical phase is a possible (but not a good) answer. Identifiers found to be too long could be truncated, or shortened, by concatenating the first N and last M letters, such that N + M is the maximum length. However, the legality of identifiers would probably be best checked in the interpretation phase, for it is in this phase that the difference between multiply-used identifiers and multiply-defined identifiers can be distinguished. Lexical cannot detect these semantic errors or errors in which block structure is involved. That is, it takes a smart lexical to recognize that these A's are different and OK:

```
DCL  1A STATIC,
        2B,
           3A   CHAR(4);
```

    However, these A's are wrong:

```
DCL  1A STATIC
        2A      CHAR(4),
        2A      CHAR(4),
```

    c. The syntactic phase could detect this. A series of reductions would be started upon encountering the IF token, and rather than calling an error routine on finding no THEN, it could assume the existence of one directly after the expression.

35. a. ⟨do stm⟩ :: = DO; | DO ⟨idn⟩ = ⟨exp⟩ TO ⟨exp⟩ BY ⟨exp⟩;

b. Reduction #

```
8a)     DO; /do/"**/7
8b)     DO ⟨idn⟩/iterative/"**/21
21)     = ⟨exp⟩ /op_prec value/"**/22
22)     TO ⟨exp⟩ /op_prec value/"**/23
23)     BY ⟨exp⟩ /OP_prec value/"**/24
24)     ;//"**/7
```

"do" puts a DO op in the matrix;
"iterative" puts a DO op in the matrix along with the indication that it is an iterative DO-group;
"value" puts the ⟨result⟩ left by op_prec into the matrix.

c. With the above approach code productions and reductions must be changed. No phases need be changed, except for the new action routines that must be added and new entries in the terminal table for DO, TO, BY.

d. In addition to the above, we must keep track of the depth of DO groups (nesting) to know which END statement is to be associated with which DO statement.

e.

```
7a)     ADD ⟨any⟩ /add/S1*/25
25)     ⟨idn⟩ TO /arg/"**/26
        ⟨idn⟩ , /arg/"**/25
26)     ⟨idn⟩ ; /final_arg/"**/7
```

"add" puts ADD op in matrix;
"final_arg" flags argument as final argument to which others are added.

36. Those containing INITIAL attributes may result in code to initialize variables indirectly, and declaration statements declaring automatic or controlled storage causes code to be generated, which assures that the appropriate storage is allocated at execution time.

37. a and b: The best view to take of all storage is that it is based storage. For example, the statement A=B (if B is static) is viewed as A - STATIC → B. Thus all accesses are preceded by loading into some register the pointer to that storage, e.g.,

```
STATIC → becomes L   15,STATIC
                 L   AC,_(0,15)
                       ↑
                       offset of B within static area
```

The statement in this problem (P1→A.B.NEXT) → A.B.C) can be interpreted as: A.B.NEXT is a pointer in a storage area and this pointer storage area has a pointer pointing to it, (i.e., P1). Therefore,

```
(P1 → A.B.NEXT) becomes  L   15,P1
                         L   13,_(0,15)
                               ↑
                               offset of A.B.NEXT within
                               A array.
```

Thus, with the pointer pointing to A.B.C now in register 13,

```
(P1 → A.B.NEXT) → A.B.C  becomes  L   AC,_(0,13)
                                    ↑
                                    offset of C within
                                    A data structure
```

Thus, solutions to both parts a and b of this question lie within the framework of the text.

38. a. For all execution time storage allocation, the compiler generates calls to the operating system. These calls are executed at the execution of the compiled source program. There is no way to know, at compile time, just what storage areas will be free at run time (indeed, they may change each time the program is loaded and run). While it might be worthwhile to maintain a table of locations for the variables in the prolog, the actual problems of assignments would be best left up to the operating system. The call may be CALL STORAGE_MANAGEMENT(POINTER,SIZE), where in POINTER the operating system returns the location of the storage area of a certain SIZE.

b. Conditions are means of handling interrupts under program control instead of using some standard system action. This involves loading locations with program entry points so that when an interrupt occurs control will transfer to the user program. In addition the ability must be reserved to take standard system action as well, transferring to the operating system and back.

   Signals are an artificial interrupt and are most easily generated by the operating system.

c. All I/O is performed by a separate computer called the I/O channel. As the reader will discover in Chapter 9, the I/O channel is a separate computer with its own instruction set. It is difficult to program this computer. Further, programming the communication between the I/O computer and the CPU is done through interrupt routines, which are tricky to write. I/O channel programs are difficult for the compiler to generate. (They are, in fact, difficult for Professor Donovan to generate.) Further, if the compiler were to generate I/O programs, it must also generate provisions for handling interrupts. Operating system subroutines are useful here as well. The compiler simply generates the appriate operating system subroutine and calls for all I/O operations.

CHAPTER 9: OPERATION SYSTEMS

1. a. A channel is basically a special purpose processor used for I/O that is subordinate to the CPU.

   b. By allowing I/O to take place totally asynchronously with program execution (unless the program is waiting for the process to be completed) throughput is increased, and the CPU is freed from the need to be interrupted for each data transfer.

   c. The channel status word contains virtually all data pertinent to the operation of the channel, as the PSW contains all data pertinent to the CPU's execution of a program. The CPU can thus check on the progress of an operation by inspecting the CSW.

   d. The channels can interrupt the CPU when an I/O operation is terminated. If transfer has been normal, then a program may have been halted waiting for I/O completion, and in the case of an abnormal condition, the operating system will have to take some kind of action, such as notifying the operator.

2. When the CPU has only such primitive I/O functions, it has to:

   a) output a "setup" command to put the device into the proper mode

   b) wait for the device's mode (status) to become favorable

   c) read or write a character

   d) go back to (b) and wait some more.

   A CPU that takes an average of 10 microseconds per instruction (on the slow side) would then sit idle 99.99% of the time. Interrupts at least allow it to accomplish something useful (execute a program) until the device tells it (without being asked) that it is ready.

3.

CCW Address	Function
1600	eject to top of next page of paper
1608	print "***** *****"
1610	print "***** *****" and skip a line
1618/1620	print "JOB NUMBER 6.251" and skip a line
1628	print "**********"
1630	print "**********" and eject to top of next page

4. Location 38: Old I/O PSW
CPU stores current PSW here when transfer upon interrupt is made to I/O routine, restores it as current PSW when I/O routine is finished.

Location 40: Channel Status Word
The channel changes this at the end of an I/O operation to show the status of the operation (normal or abnormal finish, etc.).

Location 48: Channel Address Word
User program has placed address of channel program here prior to I/O request.

Location 78: I/O New PSW
User puts location of I/O interrupt handling routine here, CPU uses it when interrupt occurs by replacing current PSW with it.

5. a. Yes. A mnemonic I/O "assembly language" could be created. Using mnemonics for instructions and symbolic addresses would be helpful and has problems similar to the regular assembler.

b. READ, WRITE, SKIP, READ CARD READER (RCR), TEST,..., etc.

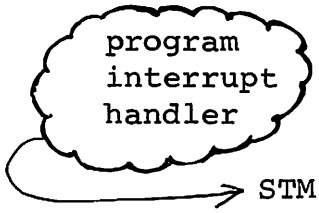
c. No basic changes are necessary. Of course the contents of the data bases are different, e.g., machine op table does not have L ~ 58 may be WRNOSPACE ~ 01.

d. Most I/O routines are very similar. The use of system macros takes the programming load from the programmer.

6. Overall strategy is to make the 360 behave like the 370. The 370 would not give a program interrupt on an instruction A 1, 1317 since the 370 does not require full word allignment for its full word data accesses. (Note: On the 370, word allignment is encouraged, for there is a high penalty for odd boundary accesses, e.g., ten times slower).

   Therefore, our strategy when encountering a A 1, 1317 is to to take the 4 bytes starting at location 1317 and place them in some properly alligned area. Then we reformulate a new instruction, A 1,DATA, in some other area and execute the new instruction. After executing that one instruction, control is then passed back to the user's program.

   Note: We will not modify the instruction in the user's program since that would make the user's program impure. The following coding of the interrupt handler is given.



For simplicity, assume this routine is located in lowest 4096 bytes and, thus, doesn't require base register.

```
          STM     1,2,SAVE  save reg 1 and 2
          L       L,44  get address from PSW
          LA      1,0(0,1)  clear first byte
          S       1,=F'4'  backtrack to interrupting instructions
          MVC     INST1(4), 0(1)  copy offending instruction
          MVC     INS2(2), 0(1)    "      "          "op code + 1
          XC      INST1(2),=X'000F'  delete opcode and R1
          OC      INST1(2),=X'4120'  convert to LA 2,...
          EX      0,INST1  compute effective address in R2
          MVC     TEMP(4), 0(2)  move data to word-allignment
                     area
          XC      INST2+1(1),=F'FO'  delete X2 from INST2
          EX      0,INST2  actually perform instruction
          LM      1,2,SAVE  restore reg 1 and 2
          LPSW    40  resume at next instruction


INST1     LA      2,0(0,0)  prototype statements (will be
                     clobbered)
INST2     A       0,TEMP       "
TEMP      DS      F  word alligned data
```

This handler will process any normal RX-type address exception (e.g., A, S, M, L, etc.) including the index and base register computation.

Note: The preceding program has the following deficiencies:

1) If any program error occurs, this interrupt routine will be executed (not just addressing errors).

2) The program will not handle store instructions.

7. Pure. b, d, and e -- these are never modified.

The symbol table, code, and matrix are all impure. If you consider an assembler (or compiler) being used in a multiprogrammed environment processing several source programs "simultaneously" it obviously cannot use any of these for all programs being processed.

8. Fragmentation exists in a multiprogrammed environment when noncontiguous blccks of storage are freed that are, individually, too small to contain another program from the queue, but that in entirety could. All methods of eliminating it involve some means of relocation. either moving the existing programs around to make room, or splitting up the next program to fit into the available areas (paging).

9.

Page	
a. has no name	has name
physical unit	logical unit
nonvariant size	variable size
(possibly more than one size, but specified)	(defined by user's needs)
invisible to user	visible to user

b. Yes. It is possible to have segmentation without paging and paging without segmentation.

10. A pure procedure is one that is non-self-modifying. That is, there is no changing of any location (data or instruction) within the procedure.

    Pure procedures are useful when more than one user is likely to be using the procedure at the same time and when re-entrant properties are desired. A third advantage of pure procedures is that they are read-only. Thus a paging environment need not write out overlayed pages.

    Advantages: 1) sharing
    2) facitates recursion
    3) read-only

11. This is a question of extremes. If a CPU is only capable of handling an average of three programs simultaneously. For instance, the use of sophisticated memory management techniques (all of which involve overhead) to keep about ten programs in core is likely to reduce overall performance. Basically, if a system is storage bound, memory management can be useful. If it is CPU bound, then it probably will not be (same for I/O bound). Most systems will not be one of these extremes, but somewhere in between.

12. In the order given, both the flexibility of address space allocation and cost in terms of complexity and overhead increase.

    a. Single contiguous allocation

       Advantages: more utilization of memory

       Disadvantage: overhead in switching users - problems - fragmentation

    b. Relocatable partitioned allocation

       Advantages: every advantage of (a) above and a solution to the problem of fragmentation

       Disadvantage: overhead in relocating partitions

c. Segmentation with demand paging

Advantages: From a memory management point of view, segmentation has the further advantage of allowing dynamic linking, which may save both memory and CPU time in that segments are not loaded unless actually called. Segmentation further allows sharing of procedures, which may save memory. Paging is another solution to the fragmentation problem.

Disadvantage: extra hardware

13. a. Informally stated, the user's address space is the memory as he sees it. Hardware devices in the system are used to map the user's address space into physical memory (and vice versa) with varying degrees of complexity and sophistication.

b. 1) Core maps into the user's address space by addition of a constant.

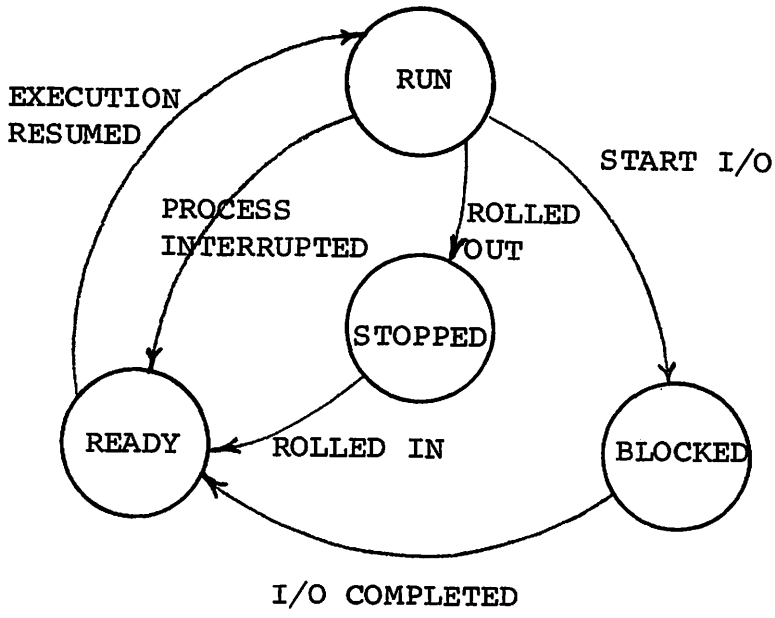
2) Same as (1) except the contant may change as programs are relocated to make room for others.

3) The user's address space is now two-dimensional; he has a number of segments, each a separate linear address space.

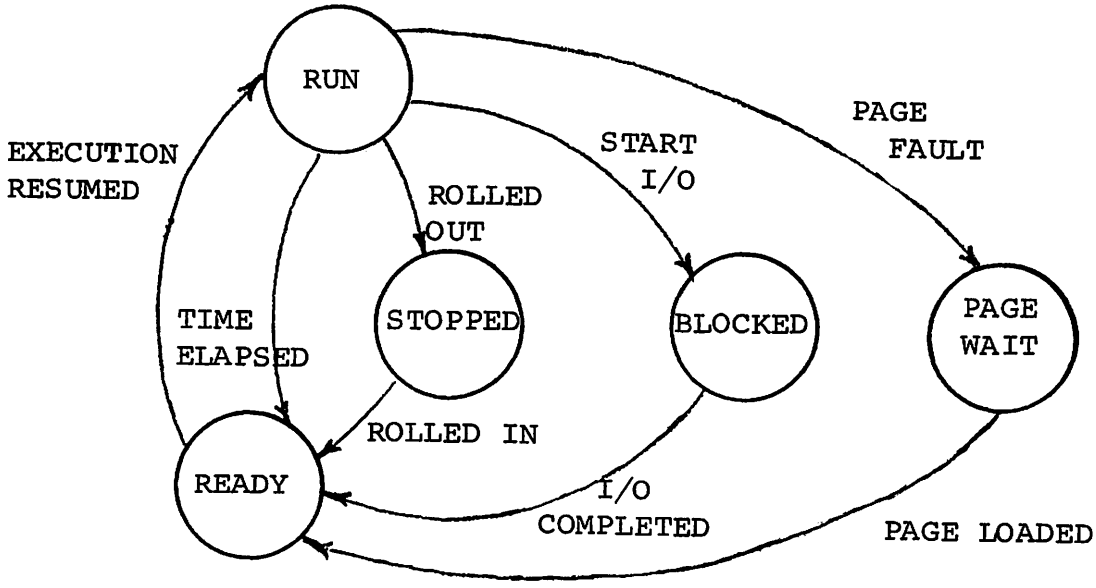
Paging breaks each segment into units, each of which maps into some section of core (by no means necessarily contiguous) by addition of a different constant for each page.

14. Everything pertinent to a process is contained in core memory (and peripheral storage). The CPU merely operates on the memory; therefore, it makes no difference which CPU it is.

15. a.

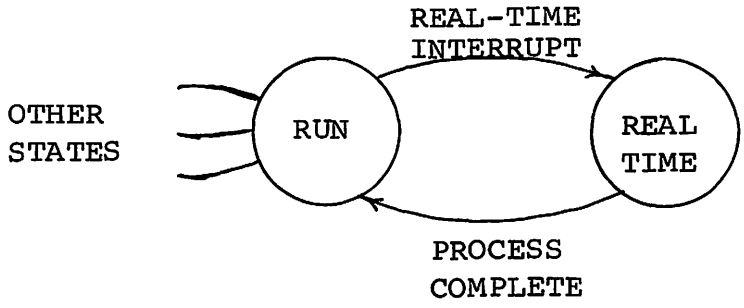


b. BLOCKED will be split.

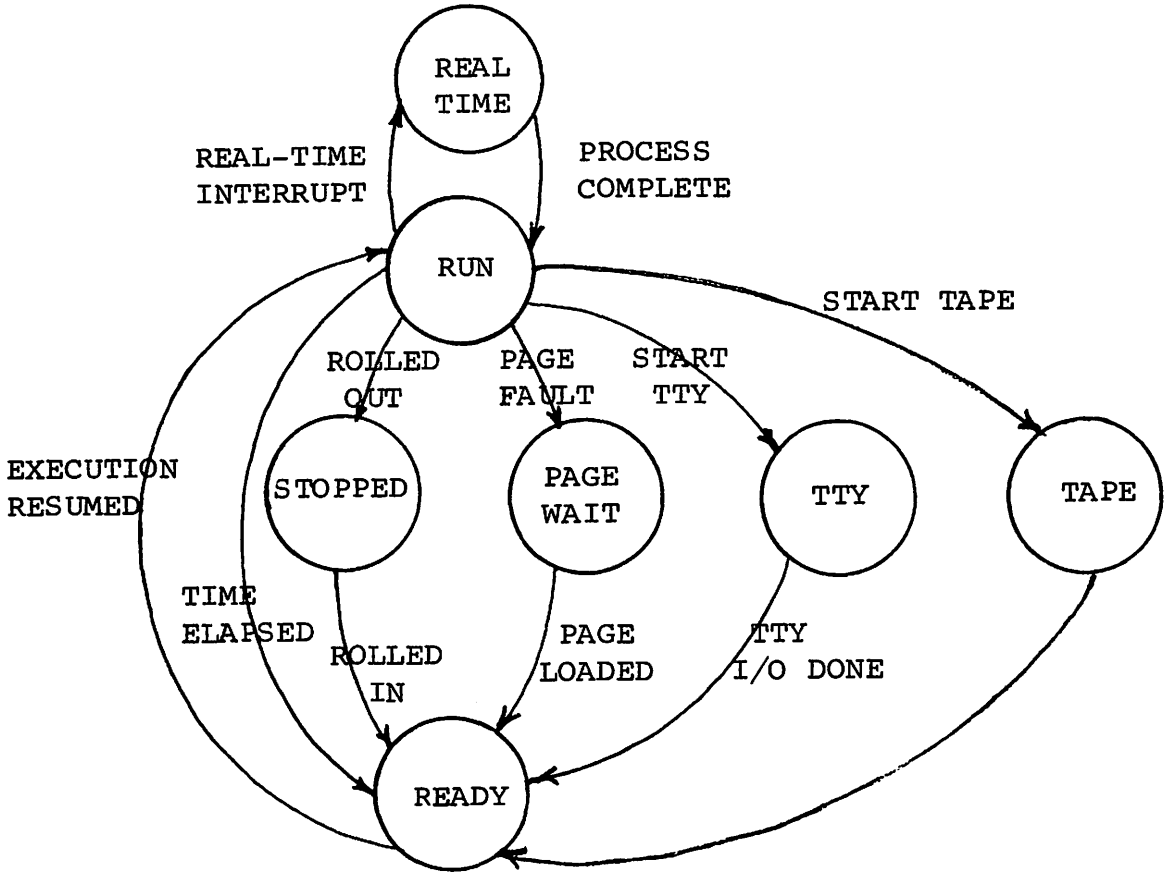


Advantages: We might want to put processes that enter READY from PAGE WAIT at the top of the ready list.

c. Add this loop:



d. TTY data transfer is several orders of magnitude slower than tape. Split BLOCKED into TTY and TAPE.



16. Races occur when two or more processes need the same device at the "same time". (See section 9.3.3). They can be resolved by making it standard practice to request devices and release them when done.

    Stalemates are races in which two processes block each other from using a needed resource. There is no apparent good solution at the present. Any ideas?

17. In a multiprocessing system, the use of software processor lockout is a form of communication between processors, to assure that two or more do not begin the same process at the same time.

    Another example is the use of I/O channels (I/O processors) and CPU, where the CPU is filling up buffer areas to be printed by the I/O channel. For efficient operation the channel may tell the CPU when it has finished printing a buffer area so that the CPU can fill it again.

18. Simultaneous Peripherals Operations On Line is a means of storing output or input on some easily available device, such as a tape drive or disc drive, as a buffer until a slower device, such as a printer, is available.

    Sometimes immediate I/O is needed, as in an emergency condition or in real-time operations, such as process control. Spooling in these instances is not reasonable.

19. The I/O traffic controller makes sure that an I/O device is used by one process at a time, and can maintain a queue of those waiting to use a device, should several processes request it while it is unavailable.

20. An example of an application suited to the use of a serial access device is the printing of a telephone directory that is in sorted form.

    An example of an application that is suited to a direct access device would be a computerized telephone information service (e.g., what is Stuart Madnick's telephone number?).

21. OPEN causes information (about a file that will be used frequently) to be copied into core. This information includes the format and location of the file to be read. If the file is to be accessed frequently, it is best not to have to access the information about the file, which is also stored on secondary storage everytime.

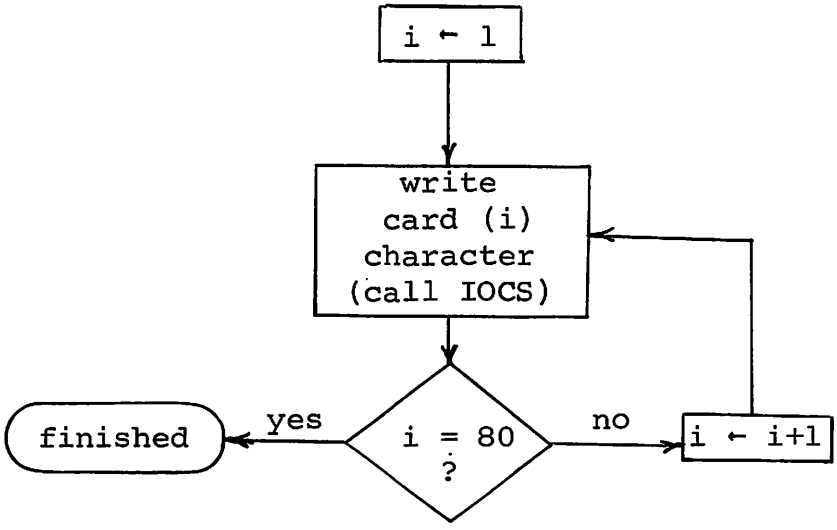
    CLOSE removes unneeded (space-consuming) information from core.

22. Segmentation provides mechanisms that would have to be included in the file system. For example, adding to files, as well as compacting files, when entries are deleted, is simplified. Protection and controlled access to programs and data is accomplished by the mechanism of segmentation.

23. a. Yes. This simple computer only affects the actual I/O mechanism, not the logical file organization.

    b. The Input/Output Control System (IOCS) and Device Strategy Module (DSM) are affected. The other modules shouldn't care.

    c. Assume sequential file system on tape. The DSM routine might look like:



24. a. Serially reusable programs can be re-executed once they are finished; they clean up after themselves or reinitialize when they begin. Another process may not use them, however, until the first process is finished. Pure procedures never change anything within themselves and can be exited and entered by any number of processes at any time. They are useful in multiprocessing and sharing purposes.

    b. No. Paging or segmentation is not necessary for timesharing. These techniques may make the operation more efficient and give additional flexibility to users.

    c. Each segment can be allocated when needed.

    d. Each segment can have a distinct class of protection associated with it.

25. a. 1) Primary storage: Core
    2) Secondary storage: Drum

    b. 1) Segmentation: File system
    Core
    DBR
    PBR
    Relocation register
    LP
    AP
    SP

    2) Paging: Disc
    Drum
    File system
    Core
    Tape
    DBR
    Relocation register

    c. Dynamic linking: File system
    Job scheduler
    Core
    PBR
    LP
    AP
    SP
    Relocation register
    PBR

d. File system
PBR
DBR
Relocation register

e. 1) File system, job scheduler, core

2) File system, job scheduler, core and file of unique file identifiers

3) File system, job scheduler, core, and file of unique file descriptors

4) File system, job scheduler, core

5) Job scheduler, drum, disc, core, extended store module

6) Same as (5) plus file system

7) Same as (5)

26. Linkage data would make a procedure impure.

27. The address of a linkage segment is contained in the linkage pointer (LP) register. Somehow the address of the linkage segment must be loaded into LP before that segment is used. One way to do this is to have the linkage segment itself load the LP with appropriate values.

28. a. READ FILE(ALPHA) RECORD(4) SIZE(55) LOCATION(BUFF);

1) Read block 0 to get first part of VTOC.
2) Search VTOC for entry ALPHA and note the block number of its file map (4).
3) Compute logical byte address [(4-1)*55 = 0165].
4) Logical byte address 0165 corresponds to logical block 0, offset 165.
5) Read ALPHA's file map (block 4) into buffer.
6) Extract physical block address for logical block 0 (block 3).
7) Read block 3 into buffer.
8) Extract bytes 165-219 and move to BUFF.

b. READ FILE(ALPHA) RECORD(19) SIZE(55) LOCATION(BUFF);

1) Read block 0 to get first part of VTOC.
2) Search VTOC for ALPHA and note block number of its file map (4).
3) Compute logical byte address [(19-1)*55 = 0990].
4) Logical byte address 0990 corresponds to logical block 0, offset 990.
5) Read ALPHA's file map (block 4) into buffer.
6) Extract physical block address for logical block 0 (block 3).
7) Read block 3 into buffer.
8) Extract bytes 990-999 and move to BUFF through BUFF+9. (This is only 10 bytes out of 55 bytes needed, thus we need next 45 bytes).
9) Read ALPHA's file map (block 4) into buffer.
10) Extract physical block address for logical block 1 (block 7).
11) Read block 7 into buffer.
12) Extract bytes 0-44 and move to BUFF+10 through BUFF+44.

29. a. The 360 has its base register facility as an aid to implementing pure procedures. It also has protection on blocks of core memory.

The 645 has indirect addressing, hardware segmentation and paging, as well as registers dedicated to the implementation of pure procedures (ap, lp, sp).

b. Yes. Segmentation is a good solution to protection. Segmentation allows logical units (instead of specific-sized units of core) to have protection of an associated class associated with them. Since all addressing is through segment table, this allows the hardware to check all accesses for legality. The hardware can perform this check by looking at the information in the segment table.

30. a. In MULTICS, a segment is loaded when referenced and some linking is done immediately thereafter (enough to satisfy the reference that caused the loading). However, references within the just-loaded segment to other segments will not be resolved until they are actually encountered.

b. There is no need for RLD cards for relocations, but some sort of RLD cards for linkage information is needed.

c. The description word would contain the location, length, type, and protection class of a segment.

31. a. The loader (linker).

b. JOHN's linkage segment.

c. The compiler that generated JOHN and its linkage segment.

d. The linkage segment of SQRT must load the LP with the proper value.

32. The "set of registers" philosophy has no real advantage other than that it may be used on a computer not designed for segmentation. It restricts the number of segments to the number of registers.

The CPU's location counter has the segment number and word number (e.g., <32,177>). But the segment number (e.g., 32) could correspond to any place in the new address space unless SWAP_DBR had the same segment number (e.g., 32) in all address spaces.

33. In each case, the remedy will be effective only if it is the limiting factor in the system as it presently exists. The reader can elaborate on examples where each suggestion could improve throughput.

34. This is left as a valuable exercise for the student or instructor.

07-017604-3