A Course in Advanced Programming for Undergraduate Computer Science Majors

by

David B. Loveman

The University of Dayton, located in Dayton, Ohio, is a "medium-sized, private, coeducational school located in the heart of the Midwest," with a full-time student body of sixty-five hundred. The University, which is also the fifth largest Catholic college in the country, "includes three schools and the college, offering a large selection of study ranging from art and philosophy to geology and computer science."

The Department of Computer Science, supported by the University's time shared RCA Spectra 70/46 and the department's own IBM 360/25, offers an interesting and challenging undergraduate program. Computer Science students are offered a wide selection of courses, both technical and non-technical, for a four year program and B.S. degree through a very flexible curriculum. Most students stress mathematics and the sciences, although a good number are interested in business management, industrial engineering, and related fields. The department offers students the opportunity for both breadth and depth equally. In addition to the formal course work, the department feels that practical, hands-on experience with real problems in the computer field is highly desirable. Students are encouraged to seek part time or summer jobs as programmers. Some jobs are available at the University computer center and other students are given the opportunity to participate in departmental research projects. Current projects include a high speed compile and go PL/I compiler for the Spectra 70/46, microprogramming extensions to the 360/25, use of an interactive vector graphic crt, and the implementation of LISP on the 70/46.

At the present time nine faculty members instruct 200 students in a total of twenty courses. The courses currently offered are detailed in Figure 1. Attention should be drawn to the use of CPS 498 to allow students to perform independent student and research under the guidance of a faculty member, and CPS 499, which is used extensively for one shot courses or to try out proposed new courses. The department actively encourages faculty and students to propose new courses to be tried as CPS 499 courses. This term's CPS 499 offerings are listed in Figure 2. Since Computer Science is a new and evolving field, the department believes that its curriculum should evolve along with the field.

Three years ago the author, a Visiting Assistant Professor of Computer Science, took over the course CPS 441-442 "Advanced Programming," known as "AP." There was little guidance as to what this two term course for junior Computer Science majors should contain. The course catalogue read "Analysis of compilers and their construction; programming techniques discussed in the current literature; advanced computer applications in both mathematical and non-numeric areas." The required prerequisite courses were a two credit course in PL/I and a three credit course in Assembly Language Programming. Some students, however, delay taking AP until their senior year. Thus the author had the problem of developing a course which would be meaningful to students who had just completed an assembly language programming course and which also would not bore those students with considerably more practical or academic experience.

Figure 1. Computer Science Courses

CPS 107	Computing-General Survey
CPS 133	FORTRAN Programming
CPS 141	ALGOL Programming
CPS 147	PL/I Programming
CPS 203	Data Processing Systems
CPS 232	COBOL Programming
CPS 245	Assembler Programming
CPS 346	Operating System
CPS 353-354	Numerical Methods
CPS 383	Logic and Set Theory
CPS 387	Logical Design
CPS 405	Computer Techniques for Business Applications
CPS 415	Introduction to Analog Computation and Simulation
CPS 416	Parallel Hybrid Computation
CPS 441-442	Advanced Programming
CPS 455-456	Numerical Analysis
CPS 481	Mathematical Logic
CPS 482	Automata Theory
CPS 498	Problems in (Named Area)
CPS 499	(Special Topics)

Figure 2. CPS 499 offerings, Spring 1972

CPS 499	Advanced Time Sharing Topics
CPS 499	Microprogramming
CPS 499	Computer Center Management
CPS 499	Interactive Graphics
CPS 499	Minicomputers

The objectives of AP are quite varied. The course strives to cover a variety of topics, looking at many superficially and a few in depth. An attempt is made to discuss practical applications and to use them as motivation for the more theoretical parts of the course. Topics such as macro processors and their implementation, searching and sorting techniques, machine architecture and microprogramming, and programming languages and their design are motivated by and discussed in a context provided by a hypothetical computer called the DAYAC.

The DAYAC, which is described in detail in the Appendix, is a machine simple enough to be easily understood yet different enough from the 360 architecture to be interesting. It is a 32 bit, word oriented machine with three general purpose registers, indexing, indirecting, and several unusual characteristics. There are three main reasons for including the DAYAC in the course: it is a simple enough machine that a student can easily understand it in detail in a short period of time, unlike most real computers; it ties the various topics of AP together and allows them to be considered as integrated in a single problem area, the DAYAC, rather than as separate, isolated topics; it allows an introduction to machine characteristics which are different than those of the 360.

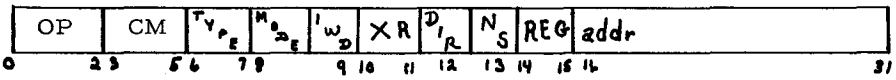
Since the course content is so broad, and since student abilities and preparation vary so widely, it is necessary to provide some form of "individualized study" within the context of the course. Since the department stresses hands on programming experience, the vehicle of a large scale programming project is used. At the beginning of a term, a project is assigned and the expected level of individual performance is given. Project assignments which have been given in the past include a simulator for the DAYAC, a cross assembler for the DAYAC Assembly Language, a version of Strachey's General Purpose Macroprocessor for use as a preprocessor for the DAYAC Assembler, and a BASIC cross compiler for the DAYAC. On completion of the projects, the best projects are incorporated into the "DAYAC system" for the use of future classes. Students are encouraged to work in teams if they so desire; they are then expected to produce correspondingly higher quality projects. Students are also encouraged to propose alternatives to the assigned project. This mechanism allows the more advanced and more ambitious to work on more interesting and challenging projects. One student became interested in the fact that the department had developed the ability to microprogram the 360/25. Available for the course, as a result of previous year's projects, are an assembler and simulator for the DAYAC, a version in PL/I to study and a version in 360 assembly language to run efficiently. This student, for his project last term, prepared an emulator, written in 360/25 microcode, for the DAYAC. Use of this emulator in future courses will make possible much more efficient use of the 360/25. Last spring the AP project was to implement a simple BASIC compiler. Three of the best students had experience implementing the department's PL/I compiler and wanted to try something more interesting. As their project they implemented a simple but operational compiler generator system. They then used this compiler generator to produce their BASIC compiler for the project.

The course content has varied from year to year in order to stress those particular topics which were especially necessary for that year's projects. In terms of the ACM's Curriculum 68, the first term is an advanced version of the course B2, Computers and Programming, with a few of the topics of I1, Data Structures. The second term is a mixture of I2, Programming Languages, and I5, Compiler Construction. Gear's book, Computer Organization and Programming, has been used quite successfully as a text for the first term, supplemented by class handouts describing the DAYAC. As yet, unfortunately, no very successful text has been found for the second term. This year we are trying Gries Compiler Construction for Digital Computers; it may prove to be too advanced for this course.

Curriculum 68 assumes that the course B2 will be both an introduction to various topics in computer science and an introduction to assembly language programming. As a result of the structure of the Computer Science Department's curriculum, students reaching AP will have completed a course in 360 assembly language programming. As a result it is possible to discuss many of the topics of B2 at a higher level than that recommended in Curriculum 68. The use of the DAYAC allows serious discussion of machine architectures that are considerably, different from the 360. The DAYAC has characteristics which make the underlying microprogrammed structure of the machine fairly clear, and its word oriented nature exposes students to new concepts precluded by sole study of the 360 architecture.

The final test of a course is how well it prepares students either for future study or for the "real world." Conversations with past students who have gone on to graduate school or who have gotten programming jobs indicates that AP has done a satisfactory job in preparing them for their future in the computer science field.

The DAYAC (DAYton Automatic Computer) is a 32 bit word oriented machine. It does arithmetic in sign-magnitude form. The DAYAC has three general purpose registers which are usable as accumulators, index registers, or memory locations. The format of a DAYAC instruction is given below:



field name	# bits	title	function
OP	3	operation code	if OP(0)=1, the operation is logical; if OP(0)=0, the operation is arithmetic
MODE	1	mode	if 0, use c(effective address) as source word; if 1, use effective address as source word (immediate addressing)
CM	3	condition mask	masks condition code to decide whether to execute instruction
DIR	1	direction	if 0, source is memory and target is register; if 1, source is register and target is memory
NS	1	negate source	if 1, negate source word before use; if arithmetic operation, change sign if logical operation, invert each bit
IND	1	indirect	if 1, indirect addressing is being used
TYPE	2	type	use is peculiar to each operation
XR	2	index register	0 value means no indexing
REG	2	operand register	0 value means use constant zero as register contents
addr	16	address	16 bit address, thus maximum memory is 64k words

The DAYAC has a "LOAD" key on the console. Pushing the LOAD key begins execution by forcing the contents of certain DAYAC registers into an initial condition to force the loading of a program. The CC or condition code is set to all ones, guaranteeing instruction execution, the IC or instruction counter is set to one, pointing at the first instruction in memory. The first three memory locations are set as follows (in hex):

1) FC800002
2) 00000004
3) 0000000A

The first instruction is on input-output instruction, using the i-o control word pair in locations 2 and 3. This i-o control word pair specifies that 10 DAYAC instructions written in hex on one card are to be read into memory starting at location 4. These 10 instructions ordinarily will be a bootstrap loader. When the two words of the i-o control word pair are executed, they are interpreted as no-op instructions. Thus the next instruction executed is the first one read in on the card.

DAL Dayac Assembly Language

instruction format: LABEL MNEMONIC OPERAND

where
- LABEL is a symbol (optional), if present it must start in column 1
- MNEMONIC is a DAYAC instruction (MR, JRI4, etc.) a DAL pseudo-op or a LABEL appearing on a preceding OPDF; it must be preceded by at least one blank
- OPERAND 1 field for pseudo-op; 1, 2, or 3 fields for real instruction; it must be preceded by at least one blank

a field is one of:

symbol	begins with letter	ABC
decimal const	begins with digit	1 2 3 8 9
hex const	begins with =	=FF10
bit const.	begins with %	%110111101

second "field" of a real instruction is a field followed by an optional modifier of the form (*), (field), or (field*)

DAL pseudo-opc are as follows:

	END	X	X is the start address of the program
	CONS	X	X is the value to be put into the word
	BSS	X	X is the number of words to be reserved
L	EQV	X	X is the value L is to be given
	ORG	X	X is the new value of the location counter
L	OPDF	X	X is the value L is to be given
	SKIP	X	Skip X lines in listing
	PAGE	X	Skip X pages in listing
	REM	X	Remark; ignore this card

Summary of 512 DAYAC Instructions

{ Move / Shift / Add / aNd / Test / Jump / Call subr / Input-output }
{ Reg is target / Mem is target } { Negate source } { Immediate }
{ 0 / 1 / 2 / 3 condition / 4 mask / 5 / 6 / 7 or -(nothing) }

Register , A { - / (*) / (R*) / (R) } , { 0 or Logical Short or - / 1 or Logical Long or Target negated or skip if Zero / 2 or Rotate Short or Result negated or skip if Not zero / 3 or Rotate Long or Result and Target or Always skip }

examples:	MR	2, A	c(2)=c(A)
	AMNI	1, A(2)	c(A+c(2))=c(A+c(2))-1
	JRI4	, A	jump to A if previous result was zero

The eight major classes of DAYAC Instructions are as follows:

name	move a full word		
mnemonic	M		
class	arith		
op	0		
summary	move source word to target location		
type modifiers	none		
condition codes	CC(0) set to 1 if source word is zero, set to 0 otherwise		
	CC(1) set to 1 if source word is positive, set to 0 otherwise		
	CC(2) set to 1 if source word is negative, set to 0 otherwise		
examples	MR	2, A	c(2)=c(A)
	MRI	3, 247	c(3)=247
	MM	2, A	c(A)=c(2)
	MMI	0, A	c(A)=0
	MRNI	3, 247	c(3)=-247
	MRN	2, A	c(2)=-c(A)

name	shift or rotate		
mnemonic	S		
class	arith		
op	1		
summary	source word contains count of number of bits to be shifted or rotated in target word. A positive count implies a left shift or rotation; a negative count implies a right shift or rotation		
type modifiers	0 or LS	logical shift, short	
	1 or LL	logical shift, long	
	2 or LS	rotate, short	
	3 or RL	rotate, long	
	note: long implies operating on a double word		
condition codes	CC(0) set to 1 if target word after shift is zero, set to 0 otherwise		
	CC(1) set to 1 if target word after shift is positive, set to 0 otherwise		
	CC(2) set to 1 if target word after shift is negative, set to 0 otherwise		
example	extract leftmost char from word in loc A		
	MMI	0,1	set reg 1 to 0
	MR	2,A	c(2)=c(A)
	SRI	1,8,LL	shift logical long left registers 1 and 2 contain two word operand effective address (8) is the shift count

name	add		
mnemonic	A		
class	arith		
op	2		
summary	source word and c(target) are added together. Result is placed in target location		
type modifiers	0 or -	do nothing	
	1 or T	negate target word before adding	
	2 or R	negate result after adding	
	3 or RT	negate target word before adding and result afterwards	
condition codes	CC(0) set to 1 if result is zero, 0 otherwise		
	CC(1) set to 1 if result is positive, 0 otherwise		
	CC(2) set to 1 if result is negative, 0 otherwise		
examples	AR	2,A	c(2)=c(2)+c(A)
	ARI	2,3	c(2)=c(2)+3
	AM	2,A	c(A)=c(2)+c(A)
	AMI	1,A	c(A)=c(A)+1
	AM	0,A,T	c(A)=-A

name	and		
mnemonic	N		
class	logic		
op	6		
summary	source word and c(target)are added together. result is placed in target location		
type modifiers	0 or -	do nothing	
	1 or T	negate target word before adding	
	2 or R	negate result after adding	
	3 or RT	negate target word before adding and result afterwards	
condition codes	CC(0) set to 1 if result is zero, 0 otherwise		
	CC(1) set to 1 if result is positive, 0 otherwise		
	CC(2) set to 1 if result is negative, 0 otherwise		
examples	NR	2,A	c(2)=c(2) .and. c(A)
	NMI	0,A	c(A)=0
	MNM	0,A,T	c(A)= .not. A
	NRN	2,A,RT	c(2)=c(2) .or. c(A)

name	test under mask and skip
mnemonic	T
class	logic
op	4
summary	source word is mask. mask is used to select bits from target word. selected bits are or'ed together and the negative of the result sets zero indicator (CC(0)). optionally skip next instruction
type modifiers	0 or - never skip 1 or Z skip if all selected bits are 0 2 or N skip if not all selected bits are 0 3 or A always skip
condition codes	CC(0) set to 1 if all selected bits are 0, 0 otherwise TR2,*+1,A test reg 2 for 0, use next word as mask CONS=FFFFFFFF this word is a constant of all ones

name	jump
mnemonic	J
class	arith
op	3
summary	jump to location specified in source word
type modifiers	none
condition codes	unchanged

examples

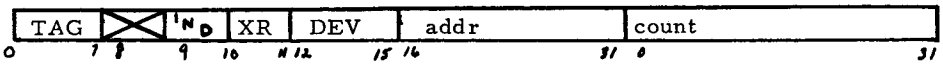
JRI	,A	unconditional transfer to loc A
JM4	2	transfer to loc contained in reg 2 if previous computation resulted in zero
AMI	0,A	add 0 to c(A), set condition codes
JR2	,B	transfer to location contained in location B if A is positive

name	Call subroutine
mnemonic	C
class	logic
op	5
summary	jump to subroutine whose location (A) is specified in source word. store IC in A, jump to A + 1
type modifiers	none
condition codes	unchanged

examples

CRI	,A	store IC in A, jump to A + 1
TMI	1,A,Z	if bit 31 of loc A is zero, skip next instruction
CRI	,B	call subr B
CM4	2	if previous computation resulted in zero, call subr. whose address is in reg 2

name	input-output
mnemonic	I
class	logic
op	7
summary	source word is first word of an i-o control word pair word with the following format (S register contains its address for use by channel program):



addr, XR, and IND are used to locate a buffer
count is number of words in buffer
DEV indicates i-o device
- 0 is card reader
- 1 is printer

tag (7) indicates direction
- 0 is read
- 1 is write

tag (6) indicates nature of data
- 0 is hex
- 1 is character

DAYAC REFERENCE MANUAL - HARDWARE

The DAYAC is a microprogrammed, stored program computer, using a subset of PL/I as its micro-machine language. A listing of the DAYAC micro program is available as a separate document. The DAYAC performs an instruction fetch and interpretation, as described below, and then executes the section of code appropriate to the particular instruction. Micro code subroutines are used to calculate the effective address, set the condition code, convert binary to hex and hex to binary, perform the sign magnitude add function, and to execute the I/O control function.

DAYAC Instruction Fetch and Interpretation

1. next instruction, as specified by contents of IC (instruction counter), is fetched and broken up, components going to the various work registers
2. IC is incremented by 1, so that it points to the next instruction
3. conditions are checked to decide whether to execute the instruction:
   DAYAC has a 3 bit CC(condition code) register:
   CC(0)=1 if previous result was zero, CC(0)=0 otherwise
   CC(1)=1 if previous result was positive, CC(1)=0 otherwise
   CC(2)=1 if previous result was negative, CC(2)=0 otherwise
   CM is used to select the bits of interest in the CC. If any of the bits selected are 1, the instruction is executed. If they are all zero, the instruction is skipped
4. the effective address of the operand in memory is calculated: add to the address the contents of register XR. if IND=0, done. If IND=1, fetch the word whose address has been calculated, get new value for address, XR, and IND and repeat.
   note: if XR=0, c(XR0=0.
5. if c(DIR)=0 c(T)=c(REG), c(S)=c(address) ie source is memory, target is register

   if c(DIR)=1 c(T)-c(address), c(S)=c(REG) ie source is register, target is memory
6. if c(MODE)=0 fetch c(S) as source word
   if c(MODE)=1 use c(S) itself as source word
7. if c(NS)=1, negate source word
   if c(OP(0))=0 operation is arithmetic, flip sign bit
   if c(OP(0̂))=1 operation is logical, flip every bit in source word
8. execute instruction: assumes source word in MDR, target address in T

DAYAC REFERENCE MANUAL - DAL, The DAYAC ASSEMBLY LANGUAGE

As yet, there is no DAL assembler written for the DAYAC itself. There is, however, a cross assembler, written in PL/I which will assemble programs written in DAL and produce DAL object decks. A listing of the DAL cross assembler is available as a separate document.

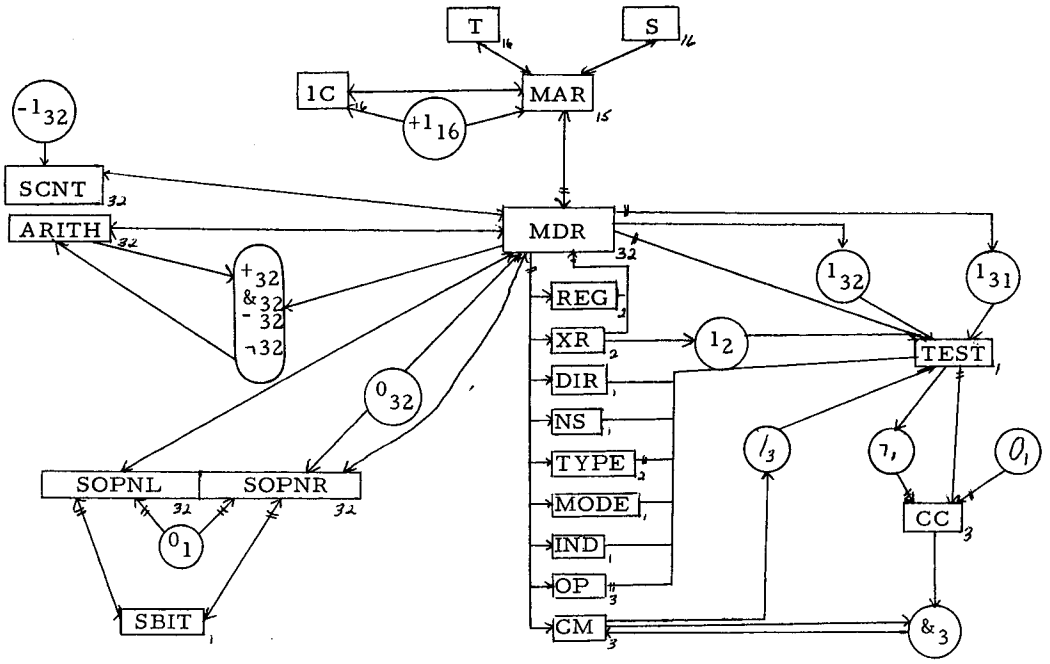
The DAL assembler is a classic two pass assembler. Pass I determines the value of each label symbol in a DAL program. Pass II produces the actual code for each instruction, with all symbolic references resolved and also prepares the object deck.

The Symbol table consists of two parallel arrays, and an index. For each symbol its external representation and 32 bit value are stored. The Operation table consists of Four parallel arrays, and an index. For each operation (machine, pseudo or OPDF) its external representation, 32 bit value, pass I branch address and pass II branch address and stored. The value must include (at least) values for the following fields: OP(0-2), CM(3-5), MODE(8), DIR(12), NS(13).

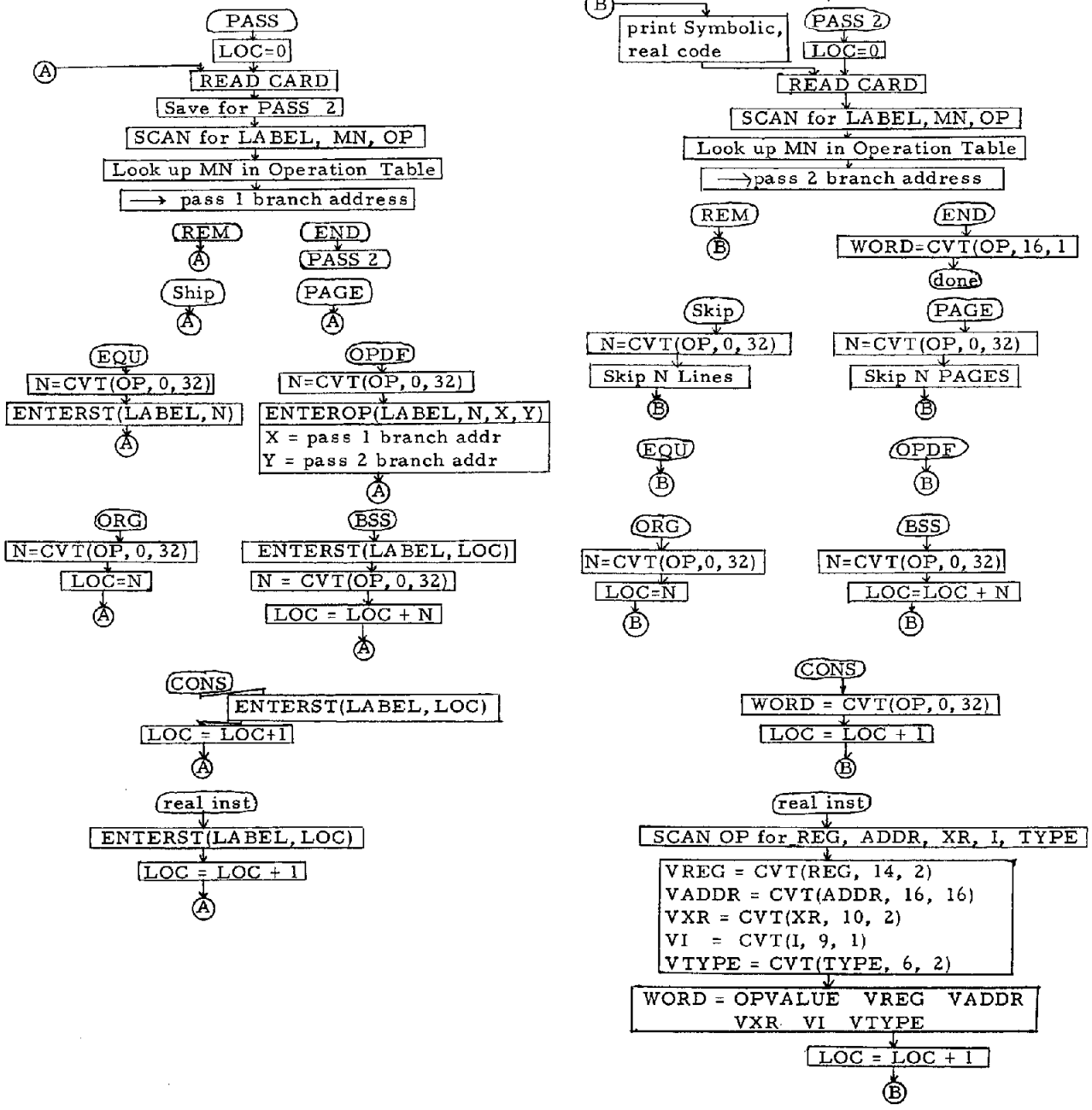
The DAL assembler uses two very powerful subroutines. The ENTER function takes an external symbol and value and enters it in the appropriate table. ENTER checks for table overflow, duplicate symbol in table, and blank symbol. The CVT function converts OP to a 32 bit value, appropriately masked and shifted. CVT(OP, I, J) returns a 32 bit value, with significant bits only in positions I thru I + J and zeros elsewhere. Value is determined: if OP is a symbol, by looking OP up on the appropriate table (error if not in table): if OP is a constant, value is calculated (binary, hex, or decimal).

The flow charts for the DAL assembler are on the following page.

The DAYAC Internal registers, the flow of data between registers, and the functions acting on registers, are depicted below: small numbers indicate data width in bits:



DAL flow charts (2 pass)



1) Bootstrap loader - The Load key causes 10 DAYAC instructions in hex to be read from a card into memory locations 4 thru 13 (hex C) and control to pass to location 4. Given below is a one card DAL program which will cause two more cards to be read in starting at location 14 (hex D). This program is more general then need be for this purpose, but merely by changing the termination constant in location 10 (hex A) this program could read in any fixed number of cards:

location (hex)	DAL instruction		comment
4	IRI	,=B	read card, control word in B
5	AM	3,=B	increase buffer address by dec 10
6	MR	2,=B	load R2 with hex buffer addr
7	ARN	2,=A	subtract termination constant
8	JRI4	,=D	if zero, done, jump to hex D
9	JRI	,4	jump to 4 to read next card
A		,=21	test word
B		,=D	input control word pair
C		,=A	

2) General loader - The two cards read in by the bootstrap loader will contain a general loader capable of reading DAL object decks. Each card of a DAL object deck has the following format:

$$\text{OnDoaaaa } X_1X_2X_3X_4X_5X_6X_7X_8X_9$$

where each Xi represents a unit of 8 hex digits or 8 blanks, n is a count indicating that the first n of the X_i's are significant, and aaaa is the address into which X_1 is to be loaded. If n is zero then aaaa is the address of the first instruction in the program to be executed. The General loader is listed below:

location (hex)	DAL instruction		comment
D	IRI	,=19	input into buffer
E	MR	1,=1C	
F	SRNI	1,24,LS	right shift 24 to isolate n
10	JR4	,=1C	if n=o, jump to addr in hex 1C
11	MR	3,=1B	1 bit indicating XR1
12	NMN	3,=1C,RT	or the bit into hex 1C
13	AMNI	1,=1C	sub 1 from hex 1C
14	MR	3,=1C(1)	load R3 with word from buffer
15	MM	3,=1C(*)	store in proper location
16	ARNI	1,1	n=n-1
17	JRI4	,=D	if n=0 read next card
18	JRI	,=14	move next word
19		,=1C	control word pair
1A		,=A	
1B		0,0(1)	hex 00100000
1C			buffer
⋮			
25			