

Lecture Notes from Manchester University Computer Science course 1970-71.

Scanned November 2011

by John Buchanan

who took the notes himself

~~END OF MARY ALMOND'S NOTES~~

MACHINE CODING

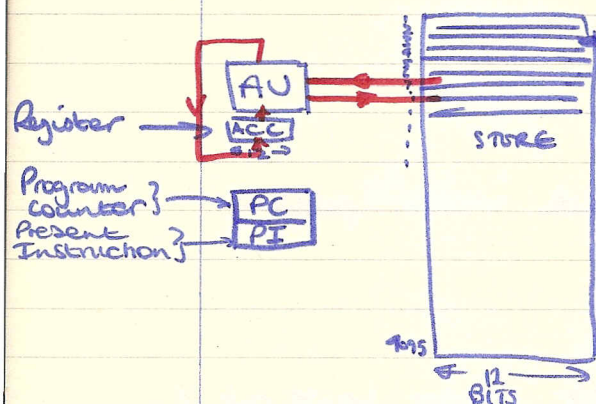
I.E. WRITING PROGRAMS IN M/C CODE LANGUAGE.

WHY?

1. Test Programs.
2. Basic System Programs e.g. Compilers, and Operating Systems.
3. Better Efficiency (for programs which are to be used many times)
4. Small Machines may not be big enough to hold a compiler

PDP 8 M/C.

Programmers view of m/c



This takes the place of a variable in AA but the store must also contain the machine code program.

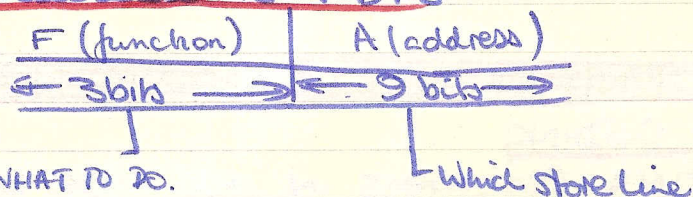
The Program Counter contains the address of the next instruction
let value of selected store line be [A]

Operation of Machine

Start on new instruction

Takes instruction from store line whose address is in "P.C."
and place in "P.I."
add 1 to 'P.C.'
decode instruction in PI
carry out action specified by instruction

Actions available to PDP8



Function List: ^{AND} 000 Acc = Acc & [A]

^{TAD} 001 Acc = Acc + [A]

^{ISZ} 010

^{DCA} 011 [A] = Acc, Acc = 0

^{JMS} 100

^{JMP} 101 PC = [A]

^{FUNNY} 110

^{VERY FUNNY} 111

Addressing System

ie meaning of A

'A' = 9 bits

Store size = 4K = 4096

A | Y | X | LINE IN PAGE

Consider the store to be divided into 2^{32} pages
each of size 128 lines

If X is 0 then goto page 0

1 then goto page in 'PC'

If Y is 0 then address is direct, then look at X

1 the address is indirect

Indirect Addressing

$$\begin{aligned} \text{ACC} &= \text{ACC} + (\text{Line 2/P.O}) \leftarrow 001000000010 \\ \text{ACC} &= \text{ACC} + [\text{LINE 3, PAGE 0}] \quad 00110000011 \end{aligned}$$

P.D.P. 8 instrumentation and controls.

On the front of the machine there are 26 switches. The left hand 6 are not used. The next 12 are the switch register and denote '1' when in the up posⁿ and '0' when down. The next 8 are 1- START, LOAD ADDR, DEP, EXAM, CONT, STOP, SINGLE STEP, SINGLE INST.

Lamps in rows of 12 are, top $\rightarrow$ bot: - Prog Count, MEMOR/ADDR, MEMOR/Buffer, ACCUMULATOR.

To load a program manually.

1. Set address of 1st instruction on switch register
Operate 'LOAD ADDR'
This sets P.C to switch register (S.R)
2. Set bit pattern of 1st instruction on S.R.
Operate 'DEP'
This copies S.R into the store line given by P.C. and adds one to P.C.
3. Set next inst on SR
Operate 'DEP'
4. etc etc.

To examine contents of store

1. Set address of 1st store line on SR
Operate 'LOAD ADDR'
This operates as 1 above

2. Operate Examue 'EXAM'

This does $ACC = [PC]$

$PC = PC + 1$

To obey Program.

CASE (A) 1 instruction at a time under operator control.

1. Set single step "UP"
2. Set single instk "DOWN"
3. Set entry address on SR
Operate "LOAD ADDR"
4. Operate "START"

This clears the accumulator and obeys one instruction and adds one to PC.

5. Operate "CONT"
this obeys one instruction and adds one to PC
Do this again etc.

CASE (B) Automatically at full machine speed

1. Set entry address on S.R
Operate "LOAD ADDR"
2. Set Single Step "UP"
Single Instk
3. Operate "START"

Simple Examples to be loaded into machine by hand

1. Sum the nos in storelines 120-127 of P1
Program in lines 0-7 of Page 1
Assume $Acc = 0$

LINE 0 / P _i	001011111000	ACC + [SL(120)] = 1370
L1	001011111001	ACC + [SL(121)] = 1371
L2	001011111010	ACC + [SL(122)] = 1372
L3	001011111011	ACC + [SL(123)] = 1373
L4	001011111100	ACC + [SL(124)] = 1374
L5	001011111101	ACC + [SL(125)] = 1375
L6	001011111110	ACC + [SL(126)] = 1376
L7	001011111111	ACC + [SL(127)] = 1377

2. Write into each store line in its own address

Program in P31

Call some work space X

Set X = -1 initially

→ ACC = X
 ACC = ^{ACC} + 1
 ACC ⇒ X
 ACC ⇒ [X]
 Jump

	binary	octal
L0/P31	001010000110	= 1206
L1	001010000111	= 1207
L2	011010000110	= 3206
L3	001010000110	= 1206
L4	011110000110	= 3606
L5	101010000000	= 5200
L6	111111111111	= 7777
L7	000000000001	= 0001

Instruction ISZ = 2

Used to program 'cycles'

ISZ 'A' [A] = [A] + 1;

Skip next instruction if [A] = 0

EXAMPLE 1

Clear ZACC

Set count = -8

Set SL 'X' = ADDR OF LINE 120

→ ACC = ACC + [X]
 X = X + 1
 ISZ 'COUNT'
 JMP
 STOP

LINE	OCTAL
0200	
0201	
0202	
0203	
0204	
0205	
0206	
0207	
0210	
0211	
0212	
0213	

0200	3215	
0201	1213	
0202	3216	
0203	1214	
0204	3215	
0205	1615	
0206	2215	
0207	2216	
0210	5205	
0211	5211	
0212	0000	
0213	7770	-8
0214	0370	ADDR OF LINE 20 P1
0215	0000	X
0216	0000	COUNT

Summary of Work so far

- 1) Instructions are 12 binary bits long
- 2) Basic instructions specify operations to be performed on 12 bits from the store (a store 'word') and in some cases a 12 bit accumulator.

3) Top 3 bits specify operation

Binary	Octal	Operation
000	0	ACC = ACC & STORE
001	1	ACC = ACC + STORE
010	2	INC STORE, SKIP IF ZERO
011	3	ACC $\Rightarrow$ STORE, ACC = 0
100	4	JUMP TO SUBROUTINE
101	5	JUMP
110	6	INPUT/OUTPUT INSTRUCTIONS
111	7	SPECIAL INSTRUCTIONS

4) Instruction splits in two parts

- a) 3 bit operation
- b) Word from store (the remaining 9 bits specify which word).

Operation Codes 6 and 7.

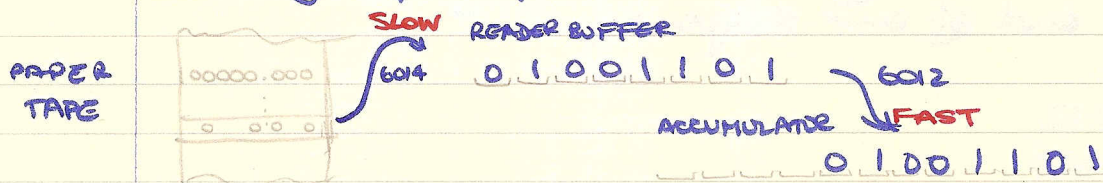
The Remaining 9 bits are not used to specify store word in these instructions.

These bits are used to specify one of many more instructions which do not require a word from store

EXAMPLES

6012	110,000,001,010	ACC = CHARACTER FROM READER BUFFER
6014	110,000,001,100	READER BUFFER = PATTERN OF HOLES ON PAPER TAPE
6011	110,000,001,001	SKIP IF READER BUFFER READY
7041	111,000,100,001	ACC = - ACC

Reading Paper Tape

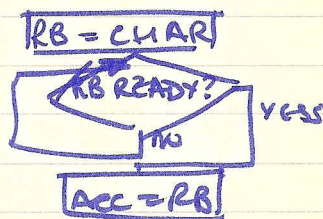


Example

- 1) Write a sequence of instrs at address 1 of some page to get a character into the accumulator.

INST. ADDR

1	6014
2	6011
3	5202
4	6012

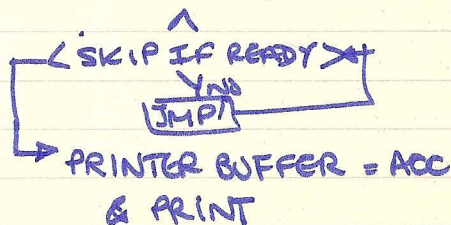


- 2) Same for Printing (at address 7 of some page)

INST. ADDR.

7	6041
8	5207
9	6046

PRINTER



We could now write such a sequence every time printing or reading was required, use of the JMS (code 4) instruction avoids this duplication

Example

INST. ADDR.

11 4200 JMS specifying word of current page

12

JMS is basic and specifies store word. The address of next instruction is stored there and we jump to the following address

$\therefore 11 + 1 = 12$ stored in word 0 jump to word 1

We have made a subroutine jump to our read a character sequence.

We get back by an indirect jump via word 0, at the end of our sequence

ADDR.

0 812

1 6014

2 6011

3 5202

4 6012

5 5600

6

7

8

9

10

11 4200

~~12~~ 4200

Operating instruction 7.

This is split into Group 1 7000
and Group 2 7400

1. 7000 No Operation
- 7100 Clear link (link is the carry at top of acc)
- 7200 Clear accumulator.
- 7040 Complement acc :- change 0's to 1's and vice versa
- 7010 moves all digits in acc + link one to the right
- 7084 " " " " " " " " " left
- 7012 " " " " " " " " " right
- 7006 " " " " " " " " " left
- 7001 Adds one to acc

2. See piece of paper.

Boot Strapping

Write simple (short) program to read other programs.
(Not very readable programs)

Write less simple prog to read other programs (more readable)
etc etc.

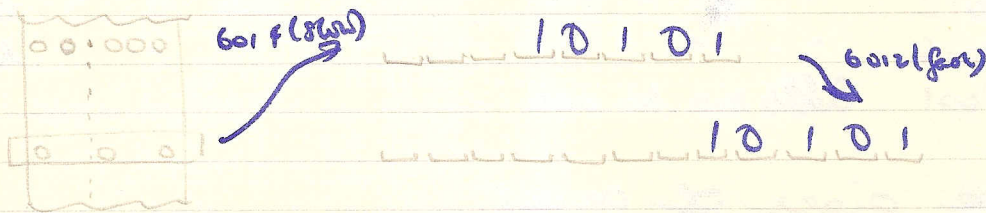
We start our bootstrap with octal loading

Conventions of Octal Loader.

- 1) Ignore all characters up to the first octal digit ^{and including}
- 2) Digit "9" marks the end of the program.
- 3) Octal digits are separated by in groups of 4 by "CR" "LF"
- 4) Tape Code 5-hole

Revision

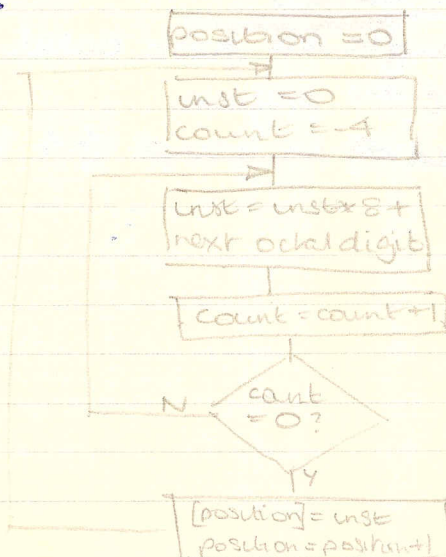
- 1) We have established a need for a program which will permit us to punch programs on paper tape rather than load them into the computer store on the handkeys.
- 2) This reading program (called an octal loader) must be short, as it will have to be loaded on the handkeys. We achieve this by having a simple form for our punched programs:- 4 octal digits per instruction.
- 3) Picture of paper tape on ~~VCR~~ is wrong, insert:-



Octal loader

First let us form groups of 4 octal digits into an instruction. We shall assume:-

- 1) A store word ~~called~~ that we shall call 'position' holds the address of another store word in which the instruction is to be placed.
- 2) The notation [position] means the content of the store word whose address is given by 'position'



17/5

PDP-8/S

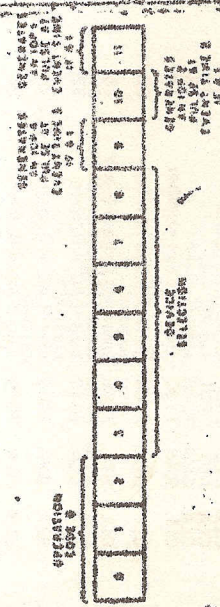
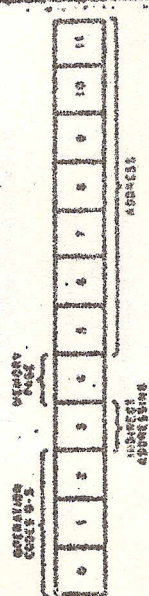
INSTRUCTION LIST

Mnemonic Code	Operation	Time (μsec)
BASIC INSTRUCTIONS		
AND	logical AND	36
OR	logical OR	36
ISZ	increment and skip if zero	44
DCA	deposit and clear AC	46
JMP	jump to subroutine	46
IOI	input/output transfer	46
OPR (I)	operate GROUP 1	236*
OPR (II)	operate GROUP 2	236
GROUP 1 OPERATE MICROINSTRUCTIONS		
NOP	no operation	26
CLA	clear AC	26
CLL	clear link	26
CMA	complement AC	26
CML	complement link	26
ROL	rotate AC and link right one	26
ROR	rotate AC and link right one	26
RTL	rotate AC and link right two	26
RLI	rotate AC and link left two	26
IAC	increment AC	26
GROUP 2 OPERATE MICROINSTRUCTIONS		
SMA	skip on minus AC	36
SZA	skip on zero AC	36
SNA	skip on plus AC	36
SNA	skip on non-zero AC	36
SZL	skip on zero link	36
SZL	skip on non-zero link	36
OSR	skip unconditionally	36
OSR	skip unconditionally	36
HLT	halts the program	36
CLA	clear AC	36

Character	Code	Character	Code
0	00000000	0	00000000
1	00000001	1	00000001
2	00000010	2	00000010
3	00000011	3	00000011
4	00000100	4	00000100
5	00000101	5	00000101
6	00000110	6	00000110
7	00000111	7	00000111
8	00001000	8	00001000
9	00001001	9	00001001
A	00001010	A	00001010
B	00001011	B	00001011
C	00001100	C	00001100
D	00001101	D	00001101
E	00001110	E	00001110
F	00001111	F	00001111
G	00010000	G	00010000
H	00010001	H	00010001
I	00010010	I	00010010
J	00010011	J	00010011
K	00010100	K	00010100
L	00010101	L	00010101
M	00010110	M	00010110
N	00010111	N	00010111
O	00011000	O	00011000
P	00011001	P	00011001
Q	00011010	Q	00011010
R	00011011	R	00011011
S	00011100	S	00011100
T	00011101	T	00011101
U	00011110	U	00011110
V	00011111	V	00011111
W	00100000	W	00100000
X	00100001	X	00100001
Y	00100010	Y	00100010
Z	00100011	Z	00100011
[	00100100	[	00100100
\	00100101	\	00100101
]	00100110	]	00100110
^	00100111	^	00100111
_	00101000	_	00101000
`	00101001	`	00101001
a	00101010	a	00101010
b	00101011	b	00101011
c	00101100	c	00101100
d	00101101	d	00101101
e	00101110	e	00101110
f	00101111	f	00101111
g	00110000	g	00110000
h	00110001	h	00110001
i	00110010	i	00110010
j	00110011	j	00110011
k	00110100	k	00110100
l	00110101	l	00110101
m	00110110	m	00110110
n	00110111	n	00110111
o	00111000	o	00111000
p	00111001	p	00111001
q	00111010	q	00111010
r	00111011	r	00111011
s	00111100	s	00111100
t	00111101	t	00111101
u	00111110	u	00111110
v	00111111	v	00111111
w	01000000	w	01000000
x	01000001	x	01000001
y	01000010	y	01000010
z	01000011	z	01000011
{	01000100	{	01000100
	01000101		01000101
}	01000110	}	01000110
~	01000111	~	01000111
?	01001000	?	01001000
@	01001001	@	01001001
A	01001010	A	01001010
B	01001011	B	01001011
C	01001100	C	01001100
D	01001101	D	01001101
E	01001110	E	01001110
F	01001111	F	01001111
G	01010000	G	01010000
H	01010001	H	01010001
I	01010010	I	01010010
J	01010011	J	01010011
K	01010100	K	01010100
L	01010101	L	01010101
M	01010110	M	01010110
N	01010111	N	01010111
O	01011000	O	01011000
P	01011001	P	01011001
Q	01011010	Q	01011010
R	01011011	R	01011011
S	01011100	S	01011100
T	01011101	T	01011101
U	01011110	U	01011110
V	01011111	V	01011111
W	01100000	W	01100000
X	01100001	X	01100001
Y	01100010	Y	01100010
Z	01100011	Z	01100011
[	01100100	[	01100100
\	01100101	\	01100101
]	01100110	]	01100110
^	01100111	^	01100111
_	01101000	_	01101000
`	01101001	`	01101001
a	01101010	a	01101010
b	01101011	b	01101011
c	01101100	c	01101100
d	01101101	d	01101101
e	01101110	e	01101110
f	01101111	f	01101111
g	01110000	g	01110000
h	01110001	h	01110001
i	01110010	i	01110010
j	01110011	j	01110011
k	01110100	k	01110100
l	01110101	l	01110101
m	01110110	m	01110110
n	01110111	n	01110111
o	01111000	o	01111000
p	01111001	p	01111001
q	01111010	q	01111010
r	01111011	r	01111011
s	01111100	s	01111100
t	01111101	t	01111101
u	01111110	u	01111110
v	01111111	v	01111111
w	10000000	w	10000000
x	10000001	x	10000001
y	10000010	y	10000010
z	10000011	z	10000011
{	10000100	{	10000100
	10000101		10000101
}	10000110	}	10000110
~	10000111	~	10000111
?	10001000	?	10001000
@	10001001	@	10001001
A	10001010	A	10001010
B	10001011	B	10001011
C	10001100	C	10001100
D	10001101	D	10001101
E	10001110	E	10001110
F	10001111	F	10001111
G	10010000	G	10010000
H	10010001	H	10010001
I	10010010	I	10010010
J	10010011	J	10010011
K	10010100	K	10010100
L	10010101	L	10010101
M	10010110	M	10010110
N	10010111	N	10010111
O	10011000	O	10011000
P	10011001	P	10011001
Q	10011010	Q	10011010
R	10011011	R	10011011
S	10011100	S	10011100
T	10011101	T	10011101
U	10011110	U	10011110
V	10011111	V	10011111
W	10100000	W	10100000
X	10100001	X	10100001
Y	10100010	Y	10100010
Z	10100011	Z	10100011
[	10100100	[	10100100
\	10100101	\	10100101
]	10100110	]	10100110
^	10100111	^	10100111
_	10101000	_	10101000
`	10101001	`	10101001
a	10101010	a	10101010
b	10101011	b	10101011
c	10101100	c	10101100
d	10101101	d	10101101
e	10101110	e	10101110
f	10101111	f	10101111
g	10110000	g	10110000
h	10110001	h	10110001
i	10110010	i	10110010
j	10110011	j	10110011
k	10110100	k	10110100
l	10110101	l	10110101
m	10110110	m	10110110
n	10110111	n	10110111
o	10111000	o	10111000
p	10111001	p	10111001
q	10111010	q	10111010
r	10111011	r	10111011
s	10111100	s	10111100
t	10111101	t	10111101
u	10111110	u	10111110
v	10111111	v	10111111
w	11000000	w	11000000
x	11000001	x	11000001
y	11000010	y	11000010
z	11000011	z	11000011
{	11000100	{	11000100
	11000101		11000101
}	11000110	}	11000110
~	11000111	~	11000111
?	11001000	?	11001000
@	11001001	@	11001001
A	11001010	A	11001010
B	11001011	B	11001011
C	11001100	C	11001100
D	11001101	D	11001101
E	11001110	E	11001110
F	11001111	F	11001111
G	11010000	G	11010000
H	11010001	H	11010001
I	11010010	I	11010010
J	11010011	J	11010011
K	11010100	K	11010100
L	11010101	L	11010101
M	11010110	M	11010110
N	11010111	N	11010111
O	11011000	O	11011000
P	11011001	P	11011001
Q	11011010	Q	11011010
R	11011011	R	11011011
S	11011100	S	11011100
T	11011101	T	11011101
U	11011110	U	11011110
V	11011111	V	11011111
W	11100000	W	11100000
X	11100001	X	11100001
Y	11100010	Y	11100010
Z	11100011	Z	11100011
[	11100100	[	11100100
\	11100101	\	11100101
]	11100110	]	11100110
^	11100111	^	11100111
_	11101000	_	11101000
`	11101001	`	11101001
a	11101010	a	11101010
b	11101011	b	11101011
c	11101100	c	11101100
d	11101101	d	11101101
e	11101110	e	11101110
f	11101111	f	11101111
g	11110000	g	11110000
h	11110001	h	11110001
i	11110010	i	11110010
j	11110011	j	11110011
k	11110100	k	11110100
l	11110101	l	11110101
m	11110110	m	11110110
n	11110111	n	11110111
o	11111000	o	11111000
p	11111001	p	11111001
q	11111010	q	11111010
r	11111011	r	11111011
s	11111100	s	11111100
t	11111101	t	11111101
u	11111110	u	11111110
v	11111111	v	11111111

Digital Equipment Corporation
MAYNARD, MASS. 01754

PRINTED IN U.S.A. 25-1/57

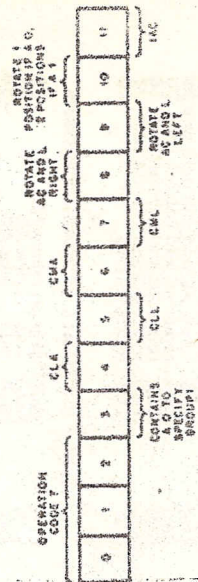


Mnemonic Code	Operation	Time (μsec)
COMBINED OPERATE MICROINSTRUCTIONS		
CIA	7041 complement AC	28
LAS	7604 load AC with switch register	38
STL	7120 set link (to 1)	28
GLK	7204 get link (out link in AC bit 1)	28
CLA	7300 clear AC and link	28
CLL	7201 set AC = 1	28
CMA	7240 set AC = -1	28
RAR	7110 shift positive number one right	28
CLL	7104 shift positive number one left	28
CLL	7106 clear link, rotate 2 left	28
RTR	7112 clear link, rotate 2 right	28
SZA	7640 skip if AC = 0, then clear AC	38
SZL	7460 skip if AC = 0, or link is 1, or both	38
SNA	7650 skip if AC ≠ 0, then clear AC	38
SMA	7700 skip if AC < 0, then clear AC	38
SMA	7540 skip if AC ≤ 0	38
SMA	7520 skip if AC < 0 or link is 1, or both	38
SPA	7550 skip if AC ≥ 0	38
SPA	7530 skip if AC ≥ 0, and if the link is 0	38
SPA	7710 skip if AC ≥ 0, then clear AC	38
SNA	7470 skip if AC ≠ 0 and link = 0	38
IOT MICROINSTRUCTIONS		
ION	6001 turn interrupt on	38
IOF	6002 turn interrupt off	38
TELETYPE KEYBOARD/READER		
KSF	6031 skip if keyboard/readers flag = 1	38
KCC	6032 clear AC and keyboard/readers flag	38
KRS	6034 read keyboard/readers buffer, static	38
KRB	6036 Clear AC, read keyboard buffer, clear keyboard flag	38

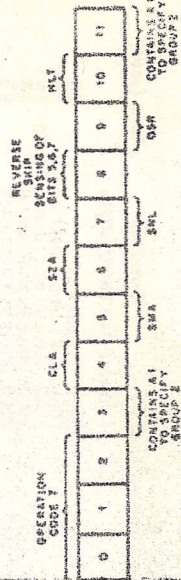
Mnemonic Code	Operation	Time (μsec)
TELETYPE TELEPRINTER/PUNCH		
TSF	6041 skip if teleprinter/punch flag = 1	38
TCF	6042 clear teleprinter/punch flag	38
TPC	6044 load teleprinter/punch buffer, select and print	38
TLS	6046 load teleprinter/punch buffer, select and print, and clear teleprinter/punch flag	38
PARITY ERROR		
SMP	6101 skip on no memory parity error	38
CMP	6104 clear memory parity error	38
HIGH SPEED PERFORATED TAPE READER TYPE PC02		
RSF	6011 skip if reader flag = 1	38
RRB	6012 read reader buffer, and clear flag	38
RFC	6014 clear flag and buffer and fetch character	38
HIGH SPEED PERFORATED TAPE PUNCH TYPE PC03		
PSF	6021 skip if punch flag = 1	38
PCF	6022 clear flag and buffer	38
PPC	6024 load buffer and punch character	38
PLS	6025 clear flag and buffer; load and punch	38

EXTENDED MEMORY CONTROL MC08

CDF	62n1	change to data field n	28
CIF	62n2	change to instruction field n	28
RDF	6214	read data field into AC 6-8	28
RIF	6224	read instruction field into AC 6-8	28
RMF	6244	restore memory field	28
RIB	6234	read interrupt buffer	28



Group 1 Operate Instruction Bit Assignments



Group 2 Operate Instruction Bit Assignments

Say $\{ 4531 \}$

INST =	000	000	000	000	
	000	000	000	100	(+4)
	000	000	100	000	(x8)
	000	000	100	101	(+5)
	000	100	101	000	(x8)
	:	:	:	:	:
	100	101	011	001	

Disadvantages

- 1) instructions must go in store word 0 onward
- 2) It doesn't ever seem to stop
- 3) It doesn't permit us to have any blank tape in our punched program.

We may improve our program to cope with these points by:-

- 1) treating first group of four octal digits as a 12 bit address - the starting address for our instructions
- 2) ignoring blank tape etc. when reading our next octal digit
- 3) Recognising the character '9' specially when reading octal digits. If we have remembered the address of the first instruction we could enter the program at this point

We shall now write octal loader program; in order to simplify it, we shall use a subroutine to form a 12 bit instruction and another to read an octal digit.

ADDRESS Octal	INSTR. Octal	Comment.
0030	4240	Jms to form instruction
0031	3010	ACC $\Rightarrow$ position, ACC = 0
0032	1010	ACC = ACC + POSITION
0033	3237	ACC $\Rightarrow$ STARTING ADDRESS, ACC = 0
0034	4240	Jms to form instruction
0035	3410	STORE INSTR. ^{or} ACC $\Rightarrow$ [POSITION]
0036	5234	JUMP TO FORM LOOP
0037	0000	STARTING ADDRESS

0040 cont'd below ↓

0041

⑤ 3410 instruction

011 100 001 000
 stor ↑ $\swarrow$ ADDR 0010
 indirect / page 0

Store words 0010 to 0017 have a special significance on the PDP computer.

If we use one of them in an indirect instruction the content has 1 added to it before the instruction is obeyed

Thus 3410 means:- 1) Add 1 to store word 0010

2) Store the acc in the store

word whose address is now in the store word 0010.

0040	0000
0041	1255
0042	3253
0043	7004
0044	7006
0045	3259
0046	4256
0047	1254
0050	2253
0051	5243
0052	5640

See
next
page

0090	0000	Return addr. stored here by JMS
0091	1255	ACC = ACC + -4
0092	3253	ACC $\Rightarrow$ COUNT, ACC = 0
0093	7004	} ACC = ACC * 8
0094	7006	
0095	3254	ACC $\Rightarrow$ INST, ACC = 0
0096	4156	JMS TO READ OCTAL DIGIT
0097	1254	ACC = ACC + INST
0056	2253	INC COUNT SKIP IF ZERO
0051	5243	JMP TO FORM LOOP
0052	5640	JMP TO ADDR STORED BY JMS
0053	0000	COUNT
0054	0000	INST
0055	7774	-4



Read Symbol

Read the next octal digit from the paper tape to the accumulator
 Ignore blank or CR & LF
 Enter program when 9 is read

An assembly language for the P.D.P. 8

This describes in convenient form a machine code program. From this description a program called an "assembler" will assemble the instructions inside the machine.

Define assembly language.

Except for control statements, each statement is equivalent to a machine instruction

```
<program> ::= <sections><enter stat.>
<sections> ::= <section> | <sections><section>
<enter stat.> ::= E<octal word>
<section> ::= <starting addr.><word sequence>
<starting addr.> ::= π<octal word><newline>
<word sequence> ::= <labelled word> | <word sequence><labelled word>
<labelled word> ::= <word><newline> | <label><word><newline>
<word> ::= <instruction> | <const>
<instruction> ::= <mnemonic><code><addr. part>
<const> ::= <octal word> | +<integer> | -<integer>
<label> ::= (<integer>)
<octal word> ::= <octal digit><octal digit><octal digit><octal digit>
<integer> ::= <decimal digit> | <integer><decimal digit>
<newline> ::= * "CR" ms "LF" * | <comment> * "CR" ms "LF" *
<comment> ::= <text>
```

SAL = Simple Assembly Language

To translate mnemonic codes we need table

AND = 0

TAD = 1

ISZ = 2

DCA = 3 etc.

In general to represent a 3 letter mnemonic code in the machine 15 bits are required

SAL imposes restrictions in order to range with 12-bits.

∴ only 16 letters ~~many~~ of alphabet may be used in mnemonic codes:-

P, H, D, T, L, R, J, Z, N, A, M, C, S, K, I

∴ in SAL

<mnemonic code> ::= AND | CLA | CLL | CMA | CML | DCA | HLT |
ISZ | IAC | JMS | JMP | RAR | RAL | RTR |
RTL | SMA | SZA | SPA | SNA | SNL | SZL |
SKP | TAD | CIA |

<addr. part> ::= <label> | <label>* | <octal digit> <octal digit>
<octal digit> | <empty>

Example of the use of SAL

Multiply Subroutine: - see sheet.

π 0200

(4) 0000

7300

CLEAR ACC AND L

DCA 003

DCA 002

TAD (1)

DCA 004

(3) TAD 001

RAR

DCA 001

TAD 002

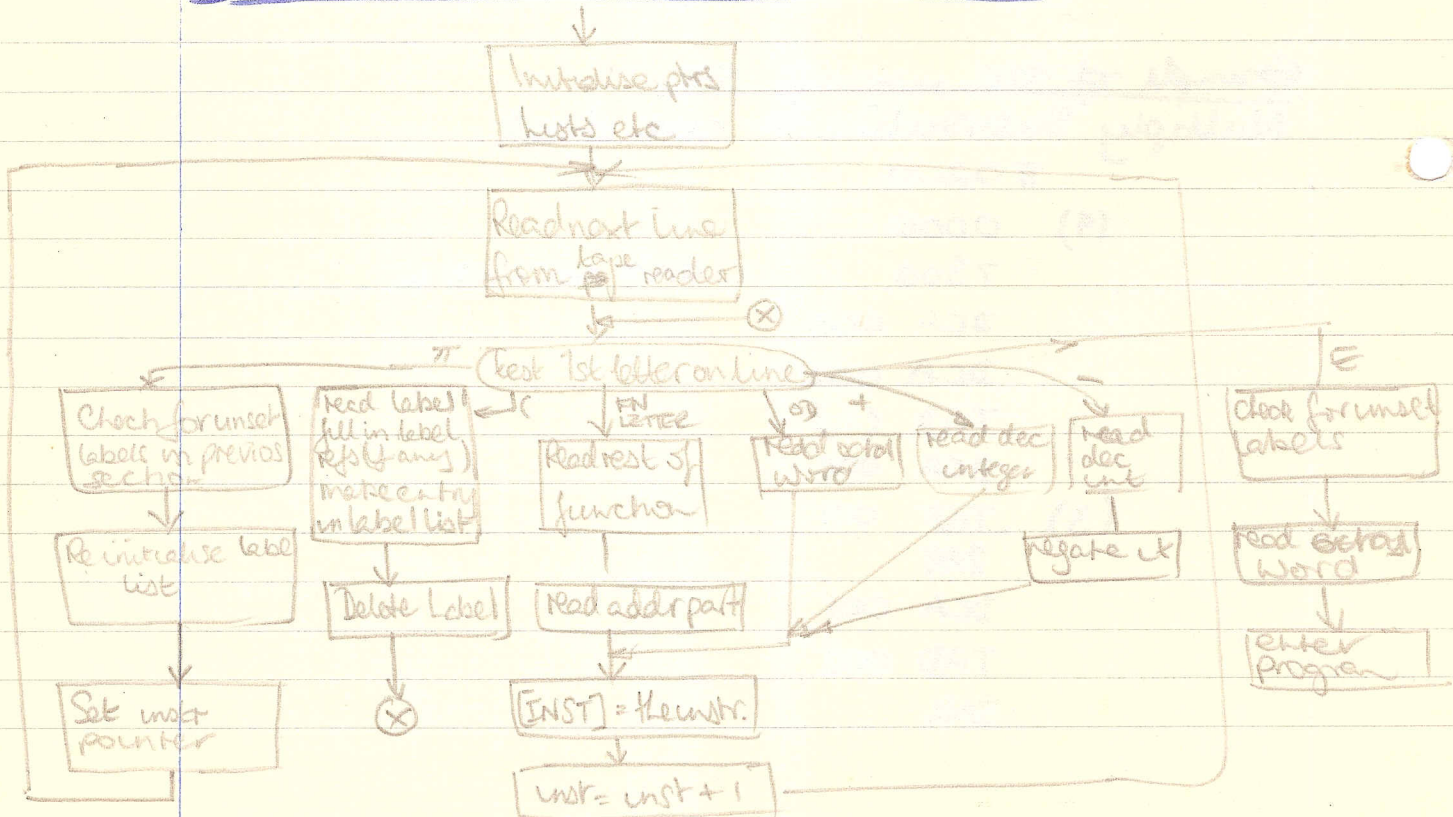
SNL

JMP (2)
 CLL
 TAD 000
 (2) RAR
 DCA 002
 TAD 003
 RAR
 DCA 003
 ISZ 004
 JMP (3)
 JMP (4)*
 (1) -12

Restrictions in SAL

- 1) A section must not cross a page boundary
- 2) Each section has its own labels which cannot be accessed from another section.
- 3) No label may be greater than 127

DESIGN OF THE SAL ASSEMBLER



READLINE (AND EDIT)Function

Read the next line of CH's from reader and ignore blanks.
Delete irrelevant CH's e.g. erase character (ER)

space

F.S. on F.S.

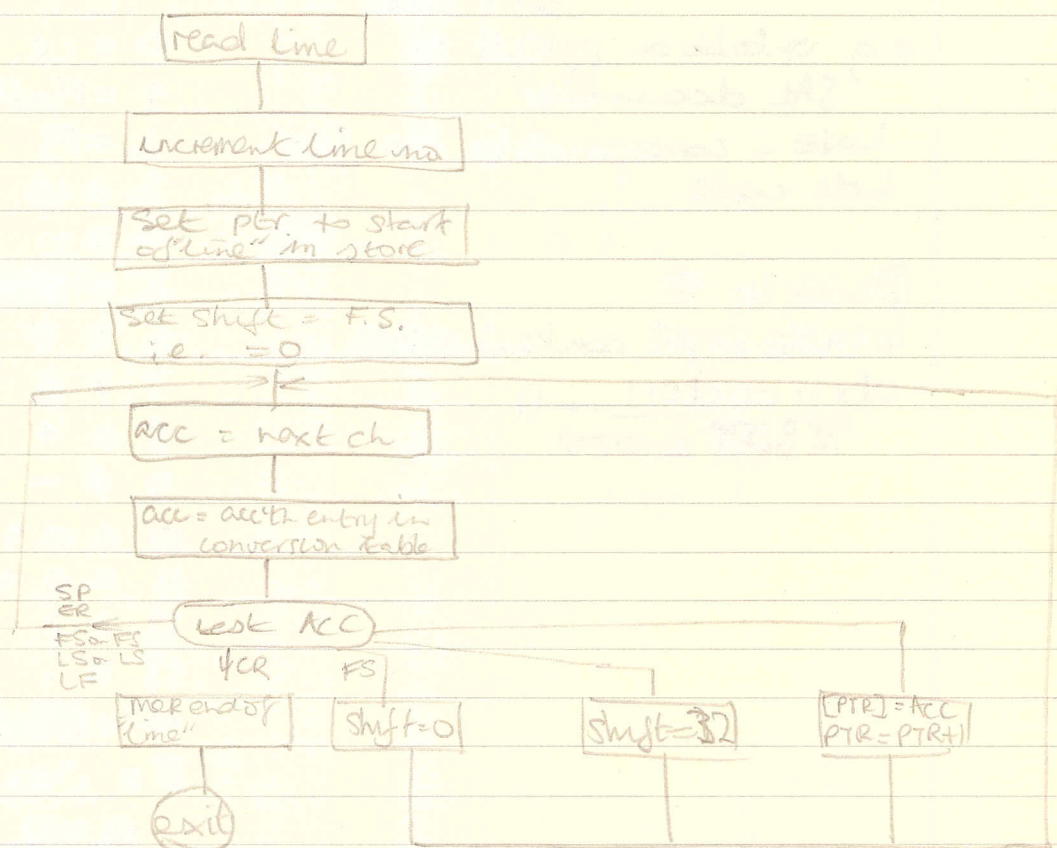
→ fig shift

L.S. on L.S.

→ letter shift.

Store the ~~the~~ line 1 character/word.

convert from 5-hole code to a more convenient form.

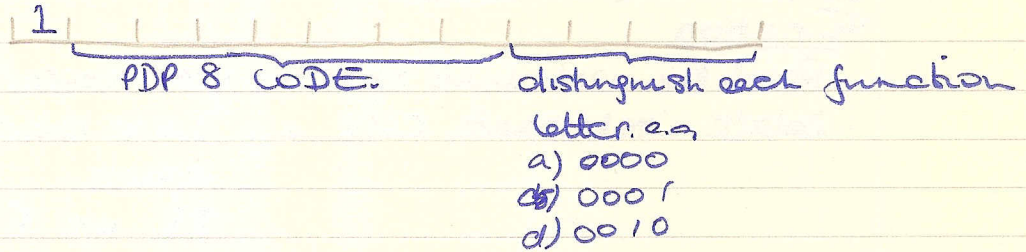


Conversion table

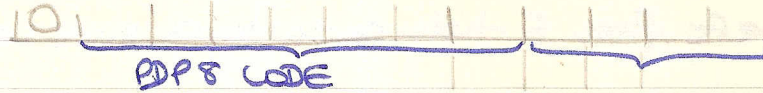
Each Entry = 12 bits

Two Cases : p.h.o

i) Function Letter.



ii) Not function letter.



e.g. on table on pp 10, 11 of
SAL document

LINE 2 corresponds to the five
hole code

Which is 8

In table on p10 contains 1613

$\equiv 01110001011 = 11$
PDP 8 code in octal $\equiv 070$

0 $\equiv$ CR

1 $\equiv$ FS on FS, LS on LS, ER, SP, LF

2 $\equiv$ FS

3 $\equiv$ LS

4 $\equiv$:

5 $\equiv$ SPARE

6 $\equiv$ π

7 $\equiv$ E

8 $\equiv$ +

9 $\equiv$ -

10 $\equiv$ 0, 4, 2, 6, 1, 5, 3, 7

11 $\equiv$ 8, 9

12 $\equiv$ |

13 $\equiv$.

14 $\equiv$ SPARE

15 $\equiv$;, ?, <, =, >, |, *, /, X, B, F, V

Q, Y, U, _, S, W, O

To initialise the program (SAL)

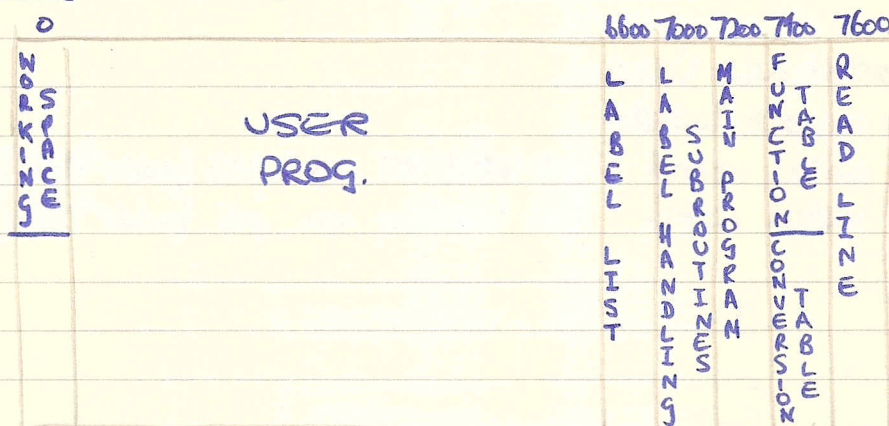
Set fault count = 0

Set line count = 0

NI = 0177

clear label list

Use of Store by SAL

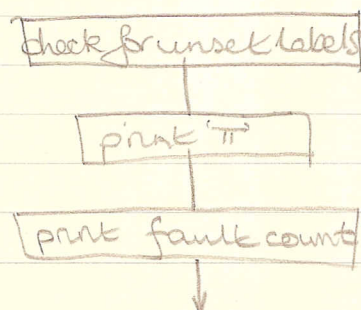


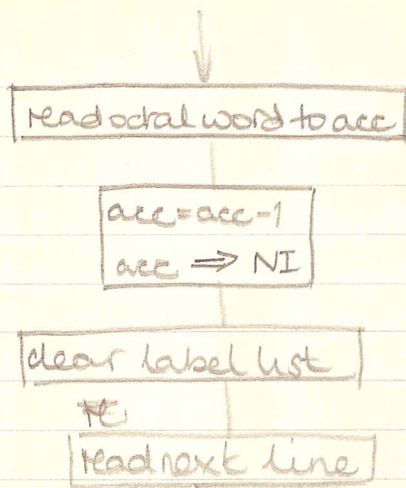
Types of line acceptable to SAL

1. π < octal word >
2. e < octal word >
3. + < decimal integer >
4. - < decimal integer >
5. < octal word >
6. < mnemonic instr >
7. (< decimal integer >) < "any of the above" >

Treatment of the above by SAL

1. π

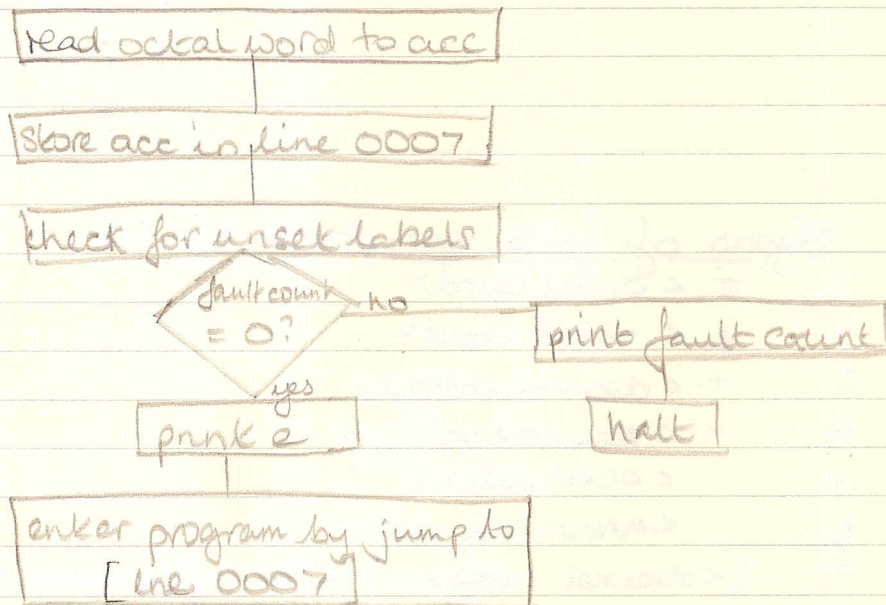




Main Subroutines Used:-

- i) count unset labels
- ii) print octal word (at label 80 on p18 (addr 7136))
- iii) read octal word (at label 40 on p5 (addr 7665))

2. e.



Main Subroutines Used:-

- i) print octal word

3 & 4.

read decimal int. into acc.

negate acc

ni = ni + 1

acc => [ni]

read next line

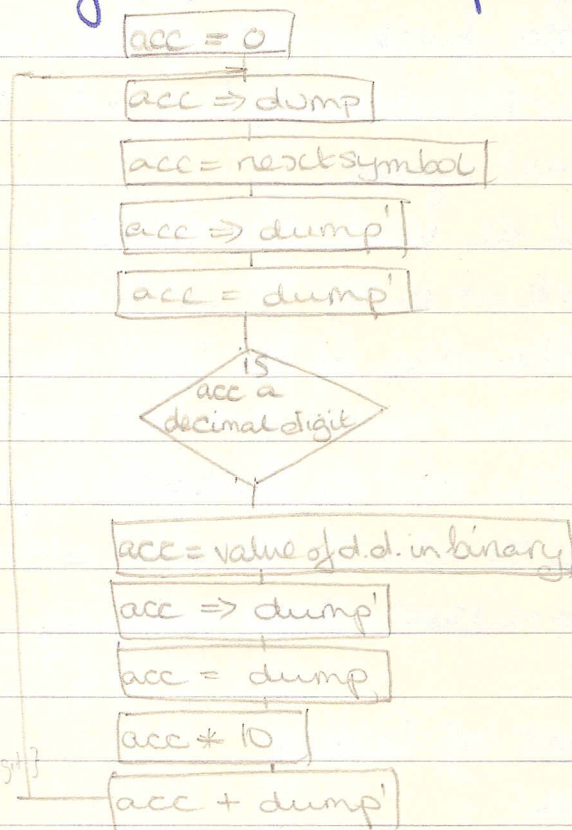
Main Subroutines Used:-

i) Read Decimal Integer (at label boon p.6 (addr 7117))

```

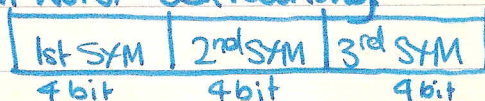
7300
(59) dca 001
    tad 416
    dca 002
    tad 002
    and (1)
    tad (61)
    sza
    jmp (62) {jmp if not}
(63) tad 002 {8,9}
    rtr
    rtr
    and (1)
    dca 002
    cll
    tad 001
    rtr
    tad 001
    ral
    tad 002
    jmp (59)
(62) lcc
    sna
    jmp (63) {jmp if}
    7300 {octal digit}
    tad 001
    jmp (60) {exit}

```



6. Action taken by sal when line starts with function letter:

i) form a 12-bit word containing



ii) Look up mnemonic function in table on page 8

iii) Take the binary machine fn. from the corresponding posⁿ on page 9.

iv) Store the function part of the instruction

v) Remember the address of this instr (in line 0003)

vi) ~~Test~~ Test next symbol

a) (: read the label and convert to binary (= L)

look at entry 'L' in label list

add the value of the label to instr

if the next symbol is * add 0400 to instr

b) octal digit: read all three octal digits and convert to binary
add the value of the octal address part into instr.

c) ~~nothing~~ neither: does nothing.

7 Treatment of labels.

The problem:

(3)	TAD 001	LABEL SET NOT PREVIOUSLY REFERENCED
	JMP (3)	LABEL REFERENCED AND PREVIOUSLY SET
	JMP (4)	LABEL REFERENCED NOT PREVIOUSLY SET
(4)		LABEL SET WHICH HAS PREVIOUSLY BEEN SET

Solution

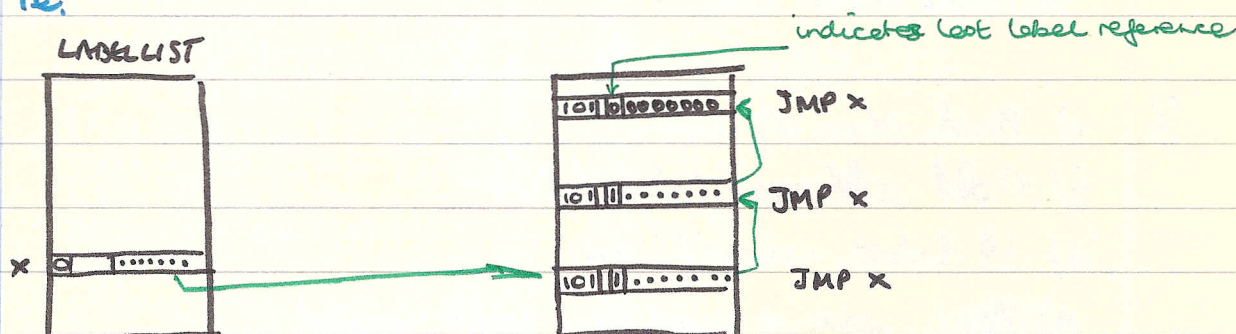
In the label list one word for each label which must be able to show: a) label set / not set
 b) label referenced / not referenced
 c) address associated with label
 d) information about instructions which reference the label before it is set.

WORD:

1 = label set
 0 = label not set

address of label

In the case of the label not set the part of the label list word used for the address is given the address of one of the label references
 i.e.

Subroutines Used.

- | | |
|---------------------------------|-------------------|
| 1. Check for unset labels | (7000 label (90)) |
| 2. Translate reference to label | (7023 " (45)) |
| 3. Set Label | (7066 " (32)) |

1. If the label keyword is true label is not set.
2.
 - i) Read value of label
 - ii) Add address of label list
 - iii) Store acc in line 000
 - iv) Add into address field of instr being translated the bottom 8 bits of ~~the~~ [line 000]
 - v) is label set
 - a) NO: form reference pointer to the instruction
 - b YES: continue
 - vi) look for '*'
 - a) '*' set indirect bit
 - b) No '*' do nothing
 - vii) Read next line.

3.
 - i) Read value of label
 - ii) Add address of label list
 - iii) Is label undefined
 - a) Yes: $acc = \text{addr of next instr.}$
 $acc \& 0177$
 $acc + 4200$
store acc in the relevant posⁿ in label list
read fr. part of instr.
 - b) No. has label been set already?
 - X: increment fault count - exit
 - N: $acc \& 0177$

Dump acc in line 001

acc = addr of next instr.

acc & 7600

acc = acc + [line 001]

dump acc in line 001

reset the link in the label list.

acc = addr of next instr (-1)

acc & 0177

acc = acc + 0201

(acc = line addr of next instr, + current page digit)

dump acc in 002

acc = ~~addr~~ 1st instr in reference chain

acc & 7400

acc = acc + 002

acc $\Rightarrow$ 1st instr in reference chain

Return to instr (iii).

PROBLEMS IN USING A PDP/8

1. Too few functions
e.g. -, *, / would be useful.
2. Too few address digits in word.
e.g. be able to address the whole store directly
3. Acc has to be dumped very often.
4. Store too small.
5. No floating point facility