PDP_8 (Program Data Processor) Digital Equipment Corporation

PDP_5 → PDP8 → 8/S → 8/I → 8/E → 8/A

(PDP8: purely transistorized; 8/I: IC; 8/A: LSI)

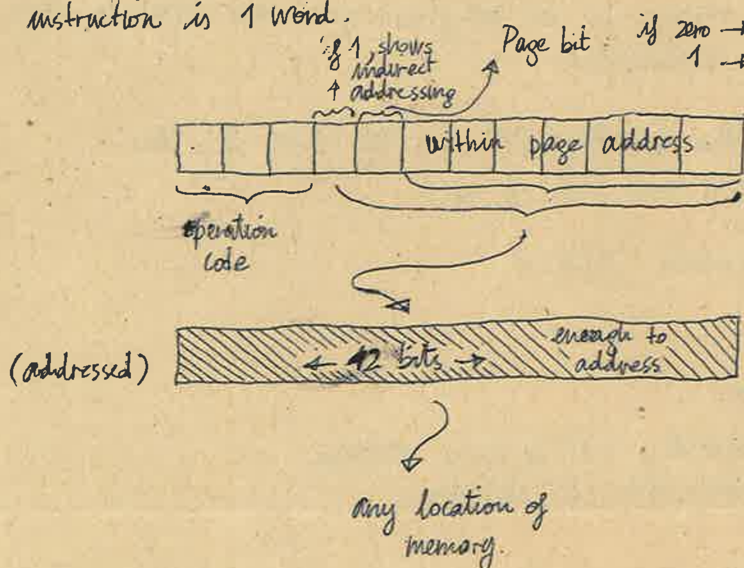
Memory in PDP-8

4096 Words (4K Words)
1K = 1024 = 2^{10}
1 Word = 12 bits

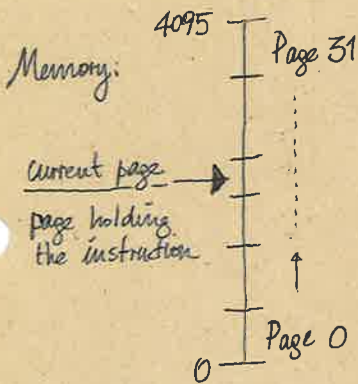
Logical organization of the 4K memory;
It is divided into pages : 32 pages having 128 words.

Instructions (Single length instructions)

Each instruction is 1 word.



Since 9 bits are not enough to address any location of memory, we use indirect addressing.



Page address : within page address

Absolute address : A 12-bit number used to adress any location in the memory.

Effective address : The address of the operand. If direct addressing is used, then it is that address. If indirect addressing is used, it is the address obtained from a directly addressed location.

In page 0, we have distinguished locations;

location 0 : holds return address for an interrupt.

location 1 : Control is transferred to here when a program interrupt happens.

locations $10_8 - 17_8$: can be used for auto-indexing when addressed indirectly (when specified by the address part of the instruction which has indirect addressing bit = 1) The effective address is not one of these locations, but it is taken from one of these locations.

Memory reference instructions :

op. code (3 bits)	I bit	P bit	7 bit page address

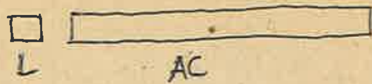
0 : AND
1 : TAD
2 : ISZ
3 : DCA
4 : JMS
6 : input output transfer instruction (IOT)
7 : operate instructions

In assembly language of PDP_8 (which is called PAL_III)

AND Y (here Y is a symbolic address), Furthermore in this case it is the effective address.

AND I Y (Y is not the effective address, but it contents are effective address.)

In PDP-8 there is a general purpose register called <u>accumulator</u> (AC). There is a single bit extension of AC on the left, to hold the overflows, called <u>Link (L)</u>.



AND Y : [(Y): shows contents of location Y]
(AC): shows contents of AC

$(Y) \wedge (AC) \longrightarrow AC$

if $(Y) = 0070_8$
and $(AC) = 3526_8$ when the above instruction is executed, AC contents will become 0020_8

TAD Y : $(Y) + (AC) \longrightarrow AC$, if there is an overflow from the leftmost bit of AC, the (L) will be complemented.

ISZ Y : $(Y)+1 \longrightarrow Y$, and if the resulting (Y) is 0_8 then skip the next instruction. (increment and skip if zero)

example 1: $(Y) = 3427_8$ then execute ISZ Y
(Y) becomes 3430_8

example 2 : You have the instructions :
ISZ Y ← when you execute, (Y) becomes 0000_8
AND Y   so next instruction is skipped.

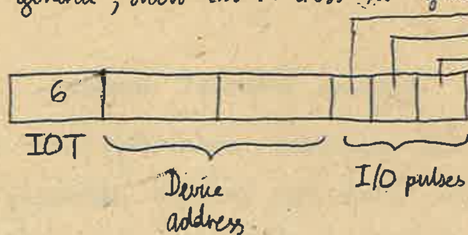
Y : 7777

DCA Y : (Deposit and clear accumulator) $(AC) \longrightarrow Y$
then $0 \longrightarrow AC$

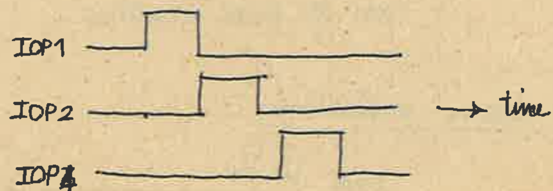
JMP Y : (JUMP) {Y} → IP (instruction pointer)
no paranthesis

JMS Y : (Jump to subroutine) $(IP)+1 \longrightarrow Y$
$Y+1 \longrightarrow IP$

IOT : In PDP-8 there are I/O devices, they have addresses 17101980
In general, such an address is given in 6 bits.



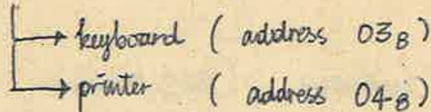
The functions in the peripheral devices of PDP-8 are implemented by sending pulses.
These are 3 :



7 : Operate instructions (for eg. CLA, rotate AC, test conditions and skip accordingly)
(displayed by remaining bits at the right)

Memory reference instructions:
- 0 - AND
- 1 - TAD
- 2 - ISZ
- 3 - DCA
- 4 - JMS } Branching instructions.
- 5 - JMP }
- 6 - IOT
- 7 - Operate instructions

TTY → Teletypewriter



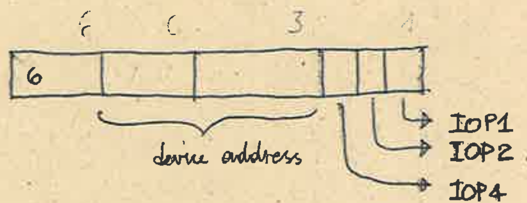
Example:

Add the contents of locations $1000_8 - 1050_8$ (inclusive) and store the result in location 377_8

```
shows 200 is an address  ← *200     / indicates beginning of a comment.
                           CLA      / Clear AC instruction.
                  BACK,    TAD I P  → first (automatically) increments the contents of P, then
                           ISZ IX     takes that as the effective address.
                           JMP BACK
                           DCA 377
                            ↓
                            ↓
                            ↓
                         *10
                        P, 777
                       IX, -50
```

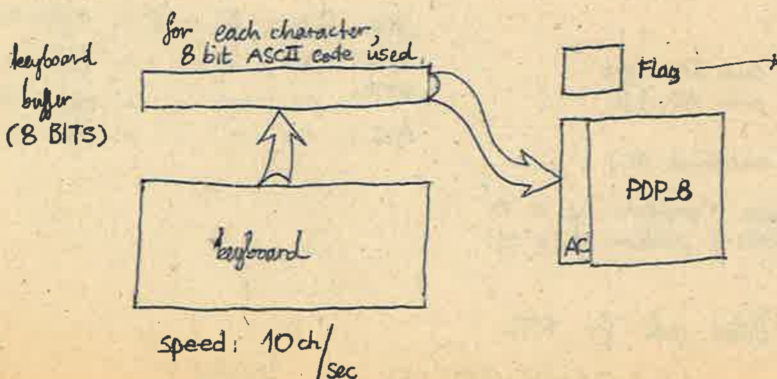
24101980

6 : I/O instructions :



instruction for the keyboard :

octal code	Mnemonic	Function (the following instruction)
6031	KSF	Skip if keyboard flag = 1
6032	KCC	Clear AC and keyboard flg.
6034	KRS	Read keyboard buffer
6036	KRB	Clear AC, keyboard flg and read keyboard buffer.



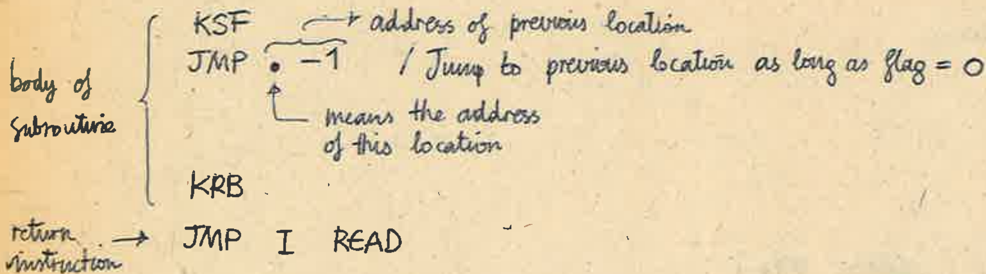
Flag → flag is used for displaying the new character is ready for computer.

Flag becomes 1 when a new character is reading (in KB buffer)

one instruction execution time: a few μSec.

A simple subprogram to read one character from the keyboard;

```
READ, 0    / will contain the return
           / address when called
```

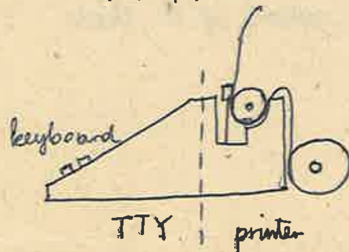


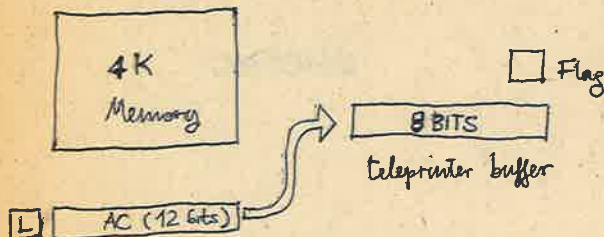
We call such a subroutine using JMS instruction. (JMS READ)

address that follows JMS is called return address.

31101980

TSF	6041	:	Skip if teleprinter flag = 1
TCF	6042	:	Clear teleprinter flag.
TPC	6044	:	Load teleprinter buffer and start printing
TLS	6046	:	TCF + TPC





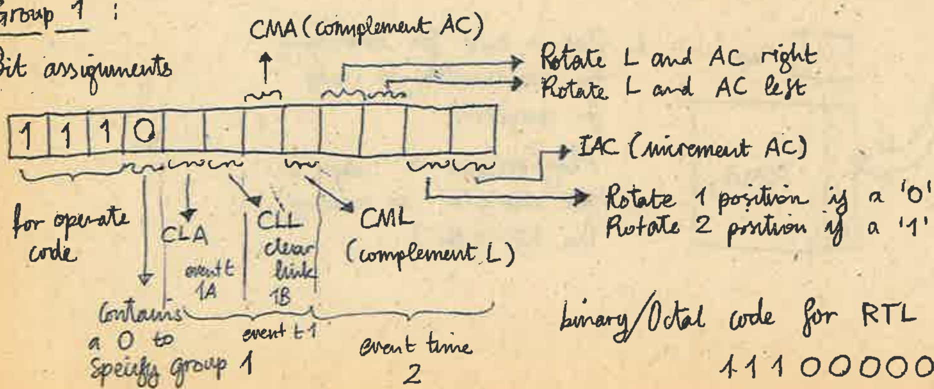
the teleprinter flag is set by circuitry when the printing of a character is completed

```
PRINT, Ø
   / Printing a character
   TSF
   JMP . -1
   TLS  / At this point the character whose ASCII code
        / is in the AC would be sent to the printer
        / for printing.
   JMP I PRINT
```

Operate instructions

Group 1:

Bit assignments



RAR : Rotate AC and L right one bits
RAL : " " " " left " "
RTR : " " " " right two "
RLL : " " " " left " "

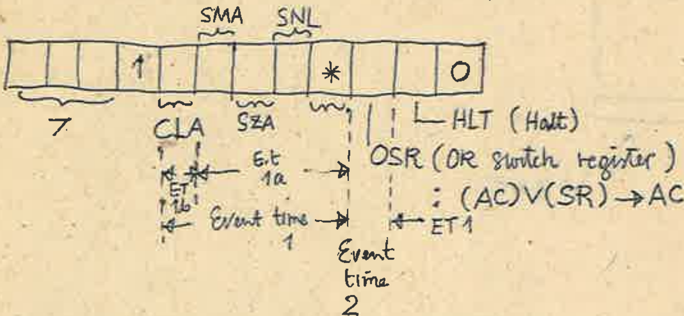
binary/Octal code for RTL :

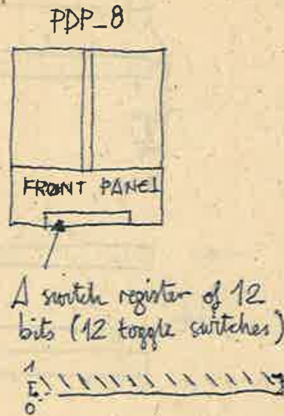
111000000110 → 7006

Group 2 Operate instructions :

there are indicate operate group 2 instructions.

Instruction code is 7000 : NOP (no operation)



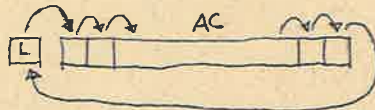


* this bit reverses the meanings of SMA, SZA, SNL

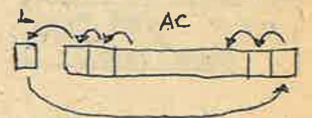
SMA : Skip on minus accumulator (i.e. if leftmost bit is 1)
SZA : " " zero "
SNL : " " non-zero link

When the Computer starts execution, it is said to be in "Run" state.
Execution of HLT, terminates this state,

Rotation to right :

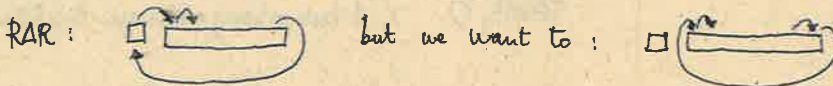


Rotation to left :



CMA ⁅ IAC → assembler OR's the codes for two symbols and places in the object program.
XXX ⁅ XXX →
CLA CLL → clear AC, clear Link
CLA IAC → clear AC, increment AC
CLL RAR

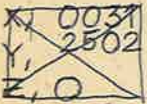
Example : Suppose we want to rotate AC right one but leave L as it is

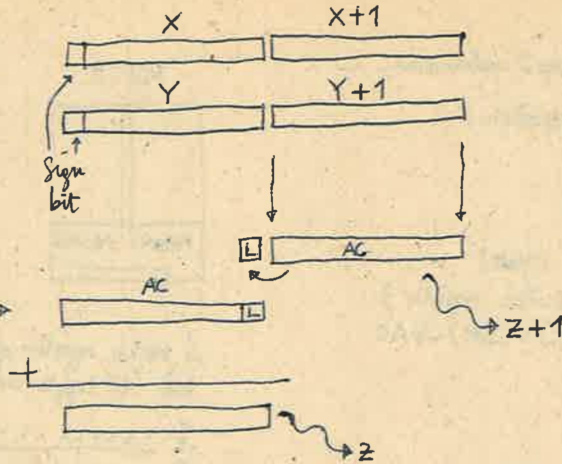



```
      *200   /A frequently used start location
      DCA  AC
      RAL
      DCA  L
      TAD  AC
      CLL⁅RAR
      DCA  AC1
      RAR
      TAD  AC1
      DCA  AC2
      TAD  L
      RAR
      TAD  AC2
      HLT
AC,0
L,0
AC1,0
AC2,0
```

Example (Addition of two 24 bit 2's complement numbers in locations X, X+1 and Y, Y+1. The results is to be in location Z, Z+1)

```
*200
  CLA CLL  →X
  TAD X+1
  TAD Y+1
  DCA Z+1
  GLK
  TAD X
  TAD Y
  DCA Z
  HLT
```






```
X, 0031
   2502
Y, 7777
   7650
Z, 0
   0
```

12111980

MT1	5th DEC	FRIDAY 1330
MT2	2nd JAN	FRIDAY 1330

→ everything until the end of this month (NOV)

```
*200   /Subtraction
  CLA
  TAD A
  CIA   /Now (AC) = -(A)
  TAD B
  HLT   /Now (AC) = (B) - (A)
A, 360
B, 521
```

Subtract contents of location A from (AC)

```
*200
  DCA TEMP
  TAD A
  CIA
  TAD TEMP
  HLT
A, 401
TEMP, 0    / A temporary storage location
```

Recommended usage of page 0:

Loc	
0	Reserve for interrupt cases.
1	" " " "
2	" " " "
3_7	You can use for constants or other purposes.
10_17	Use it if you need auto-indexing.
20_177	Store frequently used constants (ie referred to from different pages) Store variables referred to in different pages. Store subroutine references that are called from different pages.

(Page 0)

```
*20
READR, READ
TYPER, TYPES
```

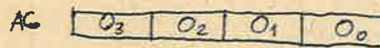
Assignment: ADD5 = TAD 0005 /1005

```
TYPE = JMS I TYPER
READ = JMS I READR
*200
 KCC }
 TLS } Make sure that TTY is initialized properly.
 TAD CHP
 TYPE
 CLA
 TAD CHD
 TYPE              / Writing 'PDP-8'
 CLA
 TAD CHP
 TYPE
 CLA
 TAD CHDASH
 TYPE
 CLA
 TAD CH8
 TYPE
 CLA
 HLT
CHP,    320   /P
CHD,    304   /D     ASCII
CHDASH, 255   /- -   Code
CH8,    270   /8
```

14111980

intelligent terminals = dumb terminal + processor
                        optal

Program for typing characters :



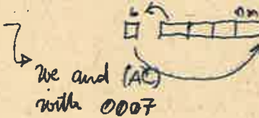

```
TYPE = JMS I TYPER

*20
TYPER, TYPES
PRAC8R, PRAC8S

*5000
TYPES, 0
    TSF
    JMP . -1
    TLS
    JMP I TYPES

*200
 / Main Program
```

```
PRAC8S, 0
   { DCA SAC    } not strongly necessary to be in the beginning
   { TAD SAC    } Storage of contents of AC into SAC
maybe RTL
in here RTL     } Rotation left 4 bits
     AND K0007  } -> We and (AC) with 0007
     TAD K260
     TYPE
     CLA
     TAD SAC
     RTL
     RAL
     DCA SAC
     TAD SAC
     AND K007
     TAD K260
     TYPE
     CLA
     TAD SAC
     RTL
     RAL
     DCA SAC
     TAD SAC
     :
```



So O_3 comes to place of O_0
and $O_2 \to O_3$
$O_1 \to O_2$
$O_0 \to O_1$

Information transfer : Character by character
Channel : I/O Processor.
Intelligent terminals : Dumb terminal + processor.

Suppose PDP-8 wants to print (AC) in octal

A : | O_0 | O_1 | O_2 | O_3 |

$(O_0+260)(O_1+260)(O_2+260)(O_3+260) \longrightarrow$ TTY

```
TYPE = JMS I TYPER
*20
TYPER, TYPES
PRAC8, PRAC8S
*5000
TYPES, Ø
    TSF
    JUMP to .-1
    TLS
    JMP I TYPES
*200
/MAIN PROGRAM
 :
 JMS I PRAC8R ...
 :
PRAC8S, Ø
    RAL
    JMS SUB
    JMS SUB
    JMS SUB
    JMS SUB
    JMP I PRAC8S
SUB, Ø
    RTL
    RAL
    DCA SAC
    TAD SAC
    AND K0007
    TAD K260
    TYPE
    CLA
    TAD SAC
    JMP I SUB
    SAC, Ø
    K260, 260
    K0007, 0007
```

PRAC8S, after finishing leaves (AC) and (L) in original form.

ASK
* Example:

Write a subroutine that OR's the contents of any specified location X at with (AC). The result being in AC and X is specified in the location following subroutine call.

* In PDP-8 to pass parameters, you can
i- use common storage
ii- " AC and L (for very simple cases)
iii- above example

```
*20
 ORR, ORS
  :
  :
 JMS I ORR
  :
  :
*5200
 ORS, 0
 CMA
 DCA TEM1
 TAD I ORS
 DCA POINTR  / POINTR contains the address of the argument
 TAD I POINTR
 CMA
 AND TEM1
 CMA    / Finished
 ISZ ORS
 JMP I ORS
```

Ex:

```
    *200
    /Main                       OR (LOCA) with (SR) put result in loc B.
200 LAS   / (SR) → AC
201 JMS I ORR
202 A  / Assembler will place 205
203 DCA B
204 HLT
205 A, 305
206 B, 0
```

```
READ = JMS I READR
*20
 READR, READS
*7100
READS, 0
    KSF
    JMP .-1
    KRB
    JMP I READS
```

```
* 3600
ASK, 0  / For entering a two digit octal number from TTY
CLA
READ
/AC contains a character code which should be in the
 range 260 – 267
  DCA OCT
  TAD OCT
  TAD M260
  SPA CLA
  HLT
  TAD OCT
  TAD M270
  SMA CLA
  HLT
  TAD OCT
  AND K0007 / 026X   → the octal digit
  RAL CLL
  RTL
  DCA D1
  READ
same [ TEST
  ↓
  TAD OCT
  AND K0007
  TAD D1
  JUMP I ASK
```

JMS ASK
/Returns with (AC) Containing the octal number which is typed by two digits

```
*3600
ASK, Ø
JMS GETDIG
JMP ERR2
RAL CLL
RTL
DCA D1
JMS GETDIG
JMP ERR2
TAD D1
COR2  ISZ ASK
ERR2, JMP I ASK
D1 , Ø
GETDIG , Ø    /gets single octal digit
CLA
READ
DCA OCT
TAD OCT
TAD M260
SPA CLA
JMP ERROR
TAD OCT
TAD M27Ø
SMA CLA
JMP ERROR
TAD OCT
AND K0007
CORECT, ISZ GETDIG
ERROR , JMP I GETDIG
OCT , Ø
K0007, 0007
M26Ø ,  -26Ø
M27Ø ,  -27Ø
*200
/MAIN
```

If you like you can put a "TYPE CR/LF" Sequence here to signal the coming of ASK

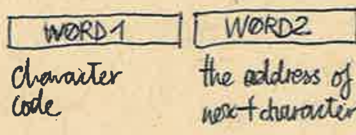
line feed – 212
Return – 215

```
AGAIN , JMS CRLF
        SKIP
        ASK
        JMS I .-1
        JMP AGAIN  /Return point in case of error
        NOP  /Return point in case of current input.
READS , Ø
KSF
JMP .-1
KRB  /Character → AC and Flag = Ø
TLS  / (AC) → Printer
JMP I READS .
*100
CRLF, Ø
  CLA
  TAD K215
  TYPE
  CLA
  JMP I CRLF
```

```
K215, 215
K212, 212
```

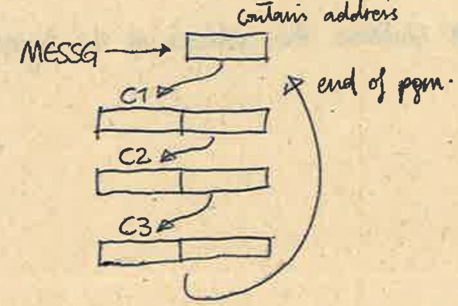
Homework TO DEC 3

For example there is a message in the memory; for each character of this message we have two words;



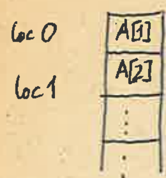
There is a location called MESSG which contains the address of the first character. The last character in the message has the address word containing MESSG.

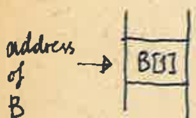
Write a PGM to print this message.



FORTRAN arrays : linear lists

Memory





B[I] is at the address :
(Address of B) + (I − 1)

linked lists

A simple linked list

link is a pointer

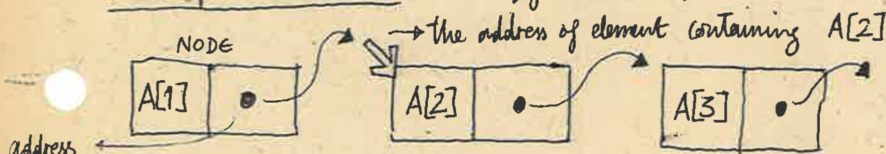
1 element of list

fields
data field
link field

In general a list consists of list elements.

A list element contains ① The contents (list item)
② Links

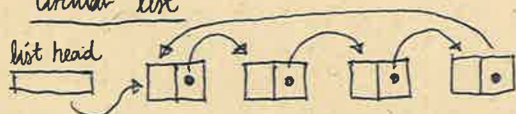
A simple linked list : (= Singly linked list)



meaning "the end"

Not organized with respect to their addresses.

Circular list



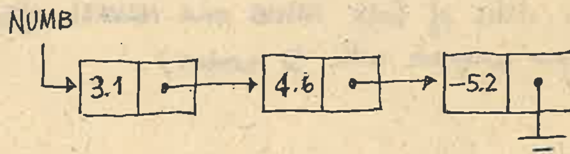
An example of linked lists : (FORTRAN)

05121980

```
      DIMENSION DATA(100), LINK(100)
      NUMB = 9
C     < Read 2.7 >
C     You want to add that to your list
C     a) Find an unused space.
C     b) Insert your number into that space
C     c) Shift list end to that space.
      NAVAIL = 1
```

	DATA	LINK	
1		2	} NODE
2		3	→ an index 1 ≤ ≤ 100
8	−5.2	0	→ represents and since it does not correspond to any index of DATA(100)
9	3.1	11	
10			
11	4.6	8	
100			

NUMB = 9 ← list head.



```
      DIMENSION  DATA(100),LINK(100)
C*    INITIALIZE THE LIST OF AVAILABLE NODES
      NAVAIL=1
      DO 10 I=1,99
10    LINK(I)= I+1
      LINK(100)=0
C*    FROM HERE ON YOU CAN BEGIN LIST MANIPULATIONS
```

```
      SUBROUTINE DELETE(LIST, DITEM)
C     THE SUBROUTINE WILL SEARCH LIST AND DELETE
C     DITEM IF IT FINDS IT
      I= LIST
90    IF(DATA(I).EQ.DITEM) GOTO 100
      IF(LINK(I).EQ.0) RETURN
      I=LINK(I)
      GO TO 90
100
```

12.12.1980

Programming Assignment: (Due 26.12.1980)

Write a FORTRAN pgm. in which there will be 20 available nodes (as a list NAVAI) and an empty list NUMB.

There will be real numbers at the input, at each step, the program will read a real number and insert it in NUMB, such that the numbers will be in ascending order. (last one is largest)

If a number to be inserted is found to be already in NUMB, this will cause that node to be freed.

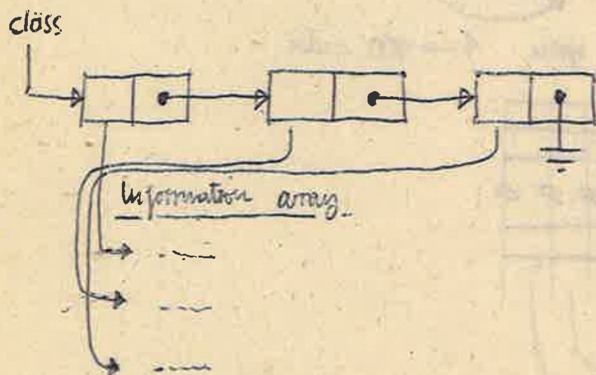
Show the states of lists NUMB and NAVAIL after each step.

(Test your program with 12 numbers)

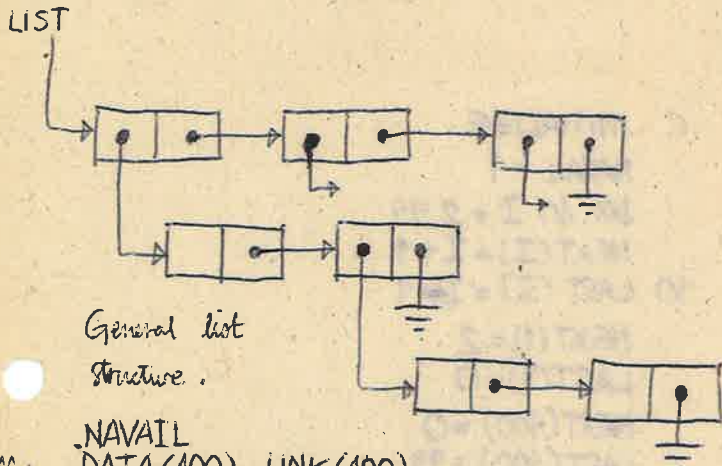
LISTS: are represented by nodes.

NODES:

Sometimes data field may contain sub-fields.
Sometimes data field may be just a pointer to a place where the data actually be stored.



Sometimes data could be structured.



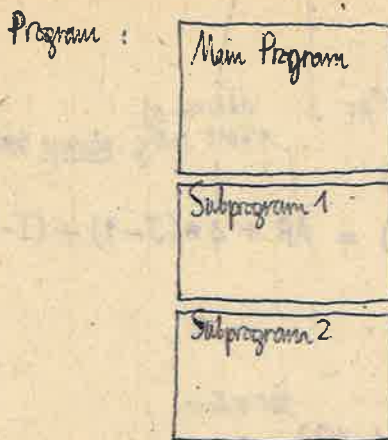

```
        NAVAIL
DIM:    DATA(100), LINK(100)
        FUNCTION NEWNOD(DUMMY)
        COMMON DATA, LINK, NAVAIL
        NEWNOD = NAVAIL
        NAVAIL = LINK(NAVAIL)
        RETURN
        END
```

→ Supplies a new node from list of available nodes.

```
        SUBROUTINE INSITE(LIST, DITEM)
C       Inserts DITEM at the beginning of LIST
        I = NEWNOD(0)
        DATA(I) = DITEM
        LINK(I) = LIST
        LIST = I
        RETURN
        END
```

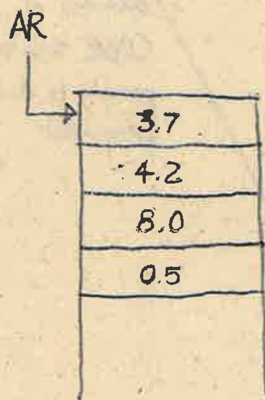
GLOBAL VARIABLES
LOCAL VARIABLES

Program:

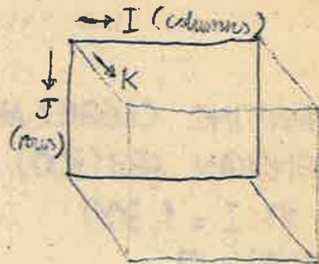


Local variables: variables which are known only in the related main or subprogram.

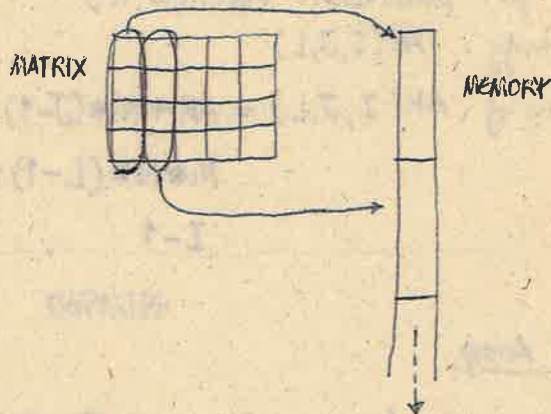
Global variables: These are known by main or subprograms, other than the one where there are defined.



AR(   )

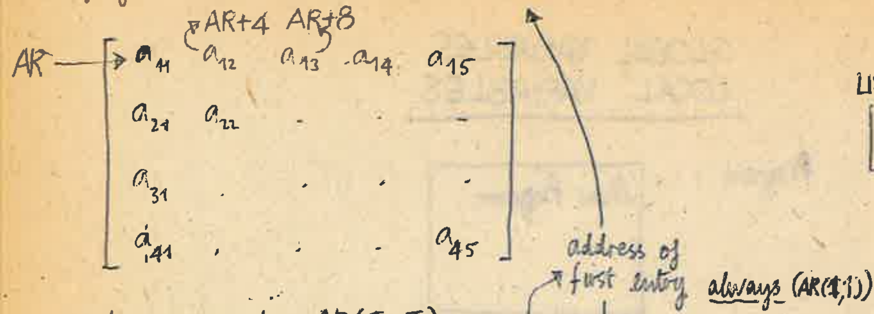


Memory can be considered as one dimensional array. In FORTRAN this mapping is done in so-called Column-major ordering.



If you have DIMENSION AR(20) and you specify AR(I) and let AR alone represent the address of AR(1) then address of AR(I) = AR + (I−1)
(This is assuming each entry occupying single location)

If you have DIMENSION AR(4,5)



and you specify AR(I,J)
then address of AR(I,J) = AR + 4*(J−1) + (I−1)

C Main Program

```
      DIMENSION CUBE(5,6,10)        300 entries
      DO 10
      DO 10                         CALL CLEAR(CUBE)
      DO 10
   10 CUBE(I,J,K) = 0
```

address of CUBE will be passed to this parameter.

```
      SUBROUTINE CLEAR(AR3)
      DIMENSION AR3(300)
      DO 10 I = 1,300
   10 AR3(I) = 0
      RETURN
      END
```

In general for DIMENSION AR(M,N,K)
and you specify: AR(I,J,L)
then address of AR(I,J,L) = AR + M*(J−1) +
M*N*(L−1) +
I−1

think!

19121980

Storage of Arrays

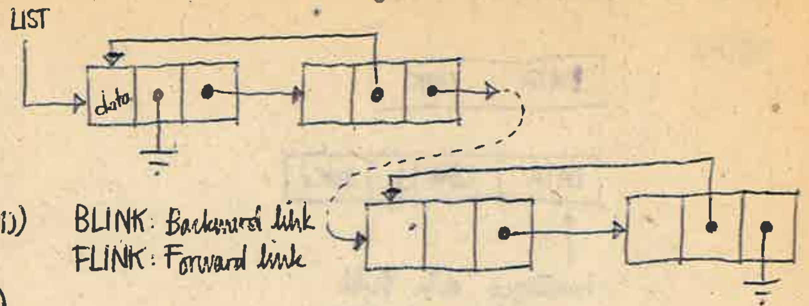
Memory. 1 dimensional (in conventional computer architectures)

AR(M,N,K):

Leftmost subscript varies most rapidly.

DOUBLY LINKED LISTS:

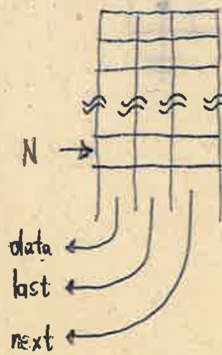
Lists with nodes having 2 links.



BLINK: Backward link
FLINK: Forward link

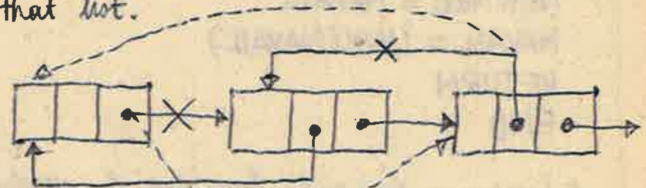
DIMENSION DATA(100), NEXT(100), LAST(100)

node space: 1 → 100 nodes.



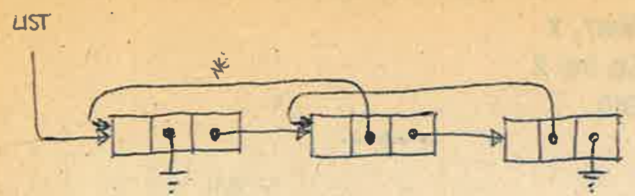

```
C     INITIALIZE
      NAVAIL = 1
      DO 10 I = 2,99
      NEXT(I) = I+1
   10 LAST(I) = I−1
      NEXT(1) = 2
      LAST(1) = 0
      NEXT(100) = 0
      LAST(100) = 99
```

Suppose we have a list "LIST"
write a subroutine to remove a node (with index N) from that list.

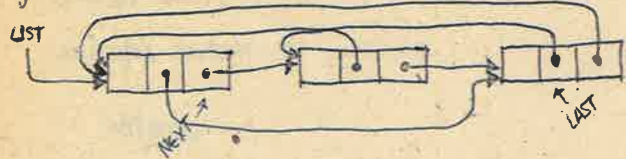


```
      SUBROUTINE REMOVE(LIST,N)     → name of the node
C     Assume that N is not the first or the last node of
C     list
      LAST(NEXT(N)) = LAST(N)
      NEXT(LAST(N)) = NEXT(N)
      RETURN
      END
```

DOUBLY LINKED LISTS :



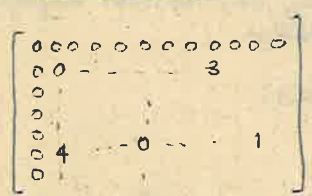
If you want to make this list circular :



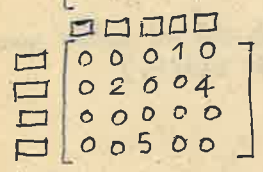
Last time we have seen a subroutine REMOVE to delete a node from a list with an assumption that the node was not the first or last node in the list. Since in a circular list, there is no first or last node in the absolute sense, REMOVE can be used to remove any node in circular list.

NEXT (LAST(NODE)) = NEXT(NODE)
LAST (NEXT(NODE)) = LAST (NODE)

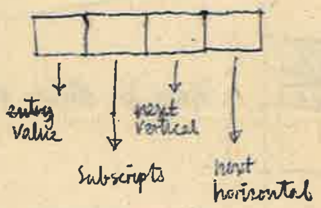
SPARSE MATRICES :

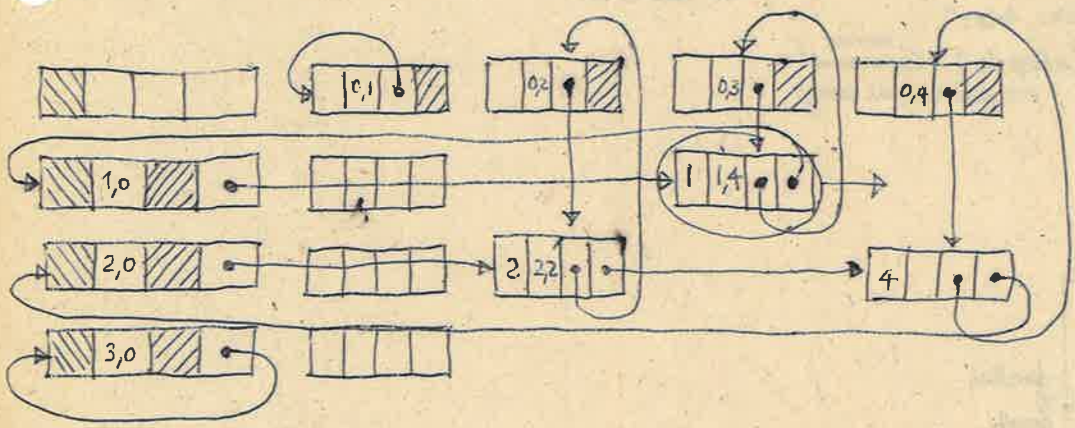


← a matrix which most of its entries are 0.



head cells (nodes) will contain meaningless data.

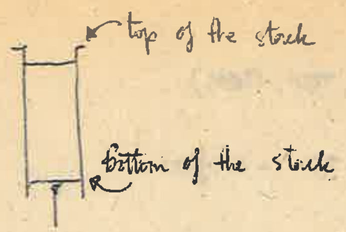




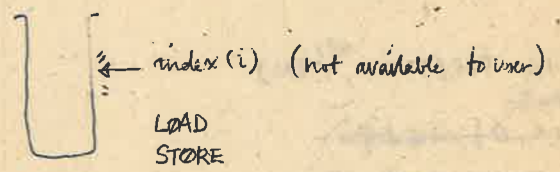
If the matrix is square,
take inverse,
transpose } exercise

stack oriented m/c → zero address m/c

STACK :

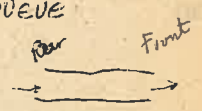


Operations : pop up an item
push down an item

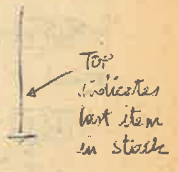


- Push-down stack : First-in, last-out stack (FILO)

- First-in, first-out (FIFO) stack : QUEUE
Add to the queue
Remove from the queue.



INTEGER STACK(100), TOP,



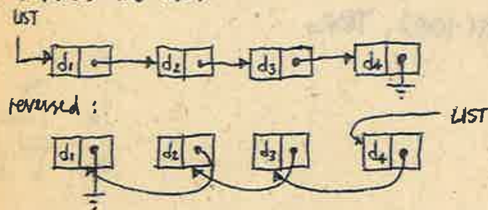
```
      INTEGER STACK(100), TOP
C     INITIALIZATION
      TOP=0

      SUBROUTINE PUSH(STACK, TOP, ITEM)
      TOP = TOP+1
      IF (TOP.GT.100) GO TO -overflow-
      STACK(TOP) = ITEM
      RETURN
      -overflow-
      RETURN
      END

      SUBROUTINE POP(STACK, TOP, ITEM)
      ~~TOP = TOP-1~~
      ~~IF (TOP.LE.0) overflow~~
      IF (TOP.EQ.0) empty stack
      ITEM = STACK(TOP)
      TOP = TOP-1
      RETURN
      END
```

31121980

Write a procedure (subroutine) REVERS(LIST) such that it reverses the list.

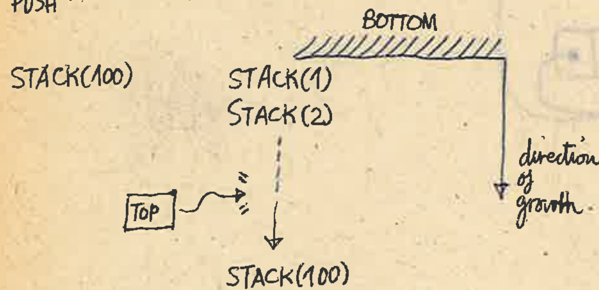


Homework due Friday.

Hierarchy of Data Structures:

Programmer simulated structures. "lists, stacks, trees"
Language Provided structures (Compiler)(interpreter) "7 dimensional" (FORTRAN)
Hardware dictated structures. (Computer) "one dimensional array"

POP
PUSH > stack operations.



Problem: Read numbers and after reading a zero, print them in reverse order.

```
      DIMENSION STACK(100)  <-- INTEGER TOP
      TOP=0
    1 READ, X
      CALL PUSH(STACK, TOP, X)
      IF(X.NE.0) GOTO 1
    2 IF(TOP.EQ.0) STOP
      CALL POP(STACK, TOP, X)
      PRINT, X
      GO TO 2
      END
```

FACT(N) → N! { 0,1 → 1 ; >1 → N[N-1]! }

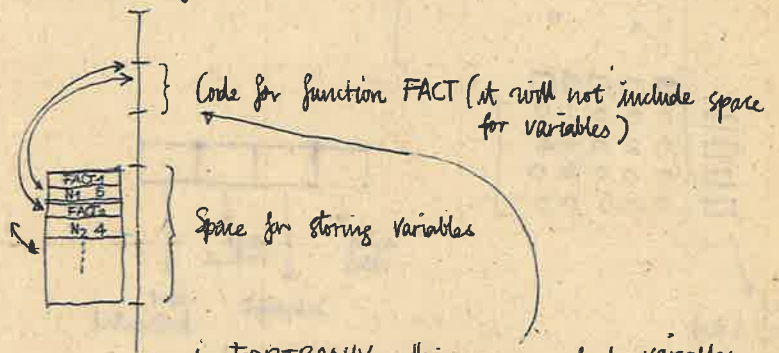
a recursive definition.

02011981

```
      INTEGER FUNCTION FACT(N)
      IF(N.LE.1) FACT=1
      IF(N.GT.1) FACT = N*FACT(N-1)      (fortunately, illegal in FORTRAN)
      RETURN
      END
```

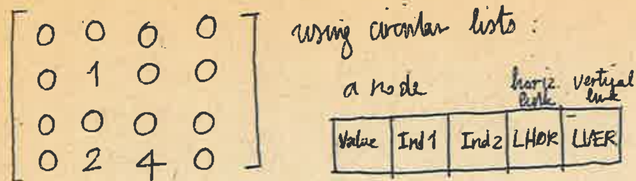
This looks like a FORTRAN subp. but contains a recursive call. Therefore, strictly speaking, it is not a FORTRAN subprogram.
So lets talk imagine an EX-FORTRAN language which allows such situations.

N=5   Memory: (EX-FORTRAN)

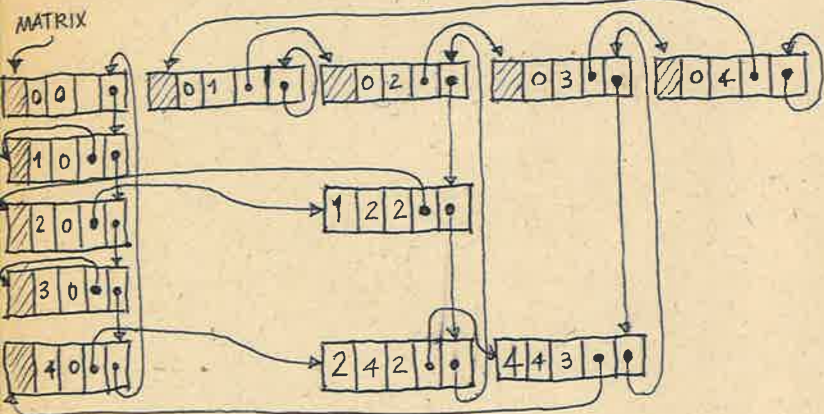


If we have value of N in activation 1, we shall call it N_1, similarly for value of N in situation 2 we use N_2 and so on.

Sparse Matrices (Revisited)



Assume that a node space is realized by the FORTRAN vectors : VALUE, IND1, IND2, LHØRZ, LVERT.




```
FUNCTION MAT(I,J)
```

This function returns the entry in the matrix corresponding to IND1 = I , IND2 = J.

```
      DIMENSION VALUE( ), IND1( ), IND2( ),
     * LHØRZ( ), LVERT( )
      CØMMØN VALUE, MATRIX
```

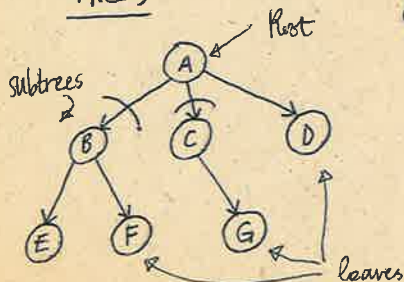
In the main program,

```
X = 3 * MAT(2,3)
```

Vector index	VALUE	IND1	IND2	LHØRZ	LVERT
1					
2					
3					
4					
5					

(name of that node → 2)

TREES



(H): a single node is also a tree (degenerated)
such a single node is also a root

If several trees are concerned, they are called a forest.
If a forest is confined under a node, ~~such that that node becomes~~, the result is a tree with that node being the root. In this case, each tree in the previous forest is called a subtree of the tree obtained.

leaves : terminal nodes. (nodes which do not have subtrees)

nodes which are not terminal nodes are called non-terminals.

degree of a node : # of subtrees connected to it.

degree of a tree : max. # of the degrees among its nodes.

level : The root is at level 1
If a node is at level l, then the roots of its subtrees are at level $l+1$.

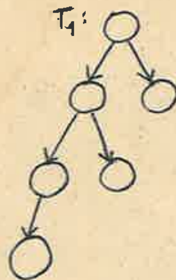
height of a tree : The max. level which exists in tree.

parents

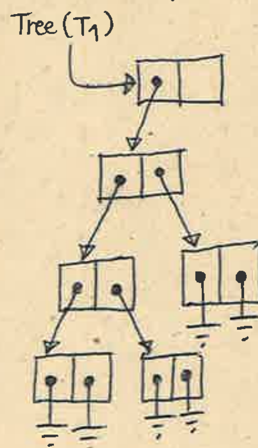
children

siblings = brothers + sisters

Binary trees : Trees with a degree at most 2.



Representation : by nodes of the sort. □□



× Exercise : Show that any Tree can be represented by a binary tree.

IBM System 360/370 Machine Language

Programs = algorithms + data structures
(some data declarations by using DC and DS assembly instructions)

Types of instructions :

1 – Arithmetic operation instructions.
2 – Logical operation instructions
3 – Branching instructions
4 – Information movement
5 – I/Ø instructions

+ control instructions used by operating system (privileged instructions)

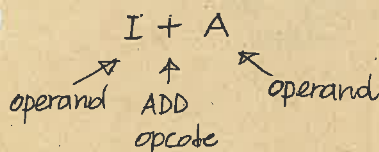
To perform the instructions there are special hardware units called as "registers" which have special purposes. Some of them may be accessed by the programmer.
the registers which are visible (usable) by the programmer ;

i) General purpose registers (GPR) (16 numbered registers : 0 1 2 3 4 5 6 7 8 9 A B C D E F)
they are called as general purpose since they are used for
- arithmetic
- addressing
- indexing (addressing)

ii) Floating point registers : (4 registers, named as 0, 2, 4, 6 ; double-word in length)

Instruction consists of two parts :

1 – operation code (opcode) : indicates the action to be done, like ADD, SUB ..
2 – operand : The action is performed on the operands specified in the instruction



in FØRTRAN ;

ISUM = ISUM + I

these variables ISUM, I are stored in memory locations in order to access them, you should know the addresses.

Operand Addressing :

If the operand is directly available in a register, to access that operand you just specify the register number.

(6) = (ISUM) ← Contents of R6 is ISUM
(5) = (I)

AR 6,5 R6 ← (R6) + (R5)

Generally operands are not directly available in registers, to access them IBM 360/370 M/C uses the following strategy:

i) (GPR) + a 12 bit number called "displacement"
here register used is called as <u>base register</u> or <u>base</u>.

ii) (GPR) + (GPR) + Displacement
(B) + (X) + D → displacement
↑ base register, ↑ index register

The summation result (the address of the operand will be created on a working register, that summation result is invisible to the programmer, since he/she cannot access to a working register.)
After the summation low-order 24 bits are used as an address
24 bits = XXXXXX
6 hex. digits.

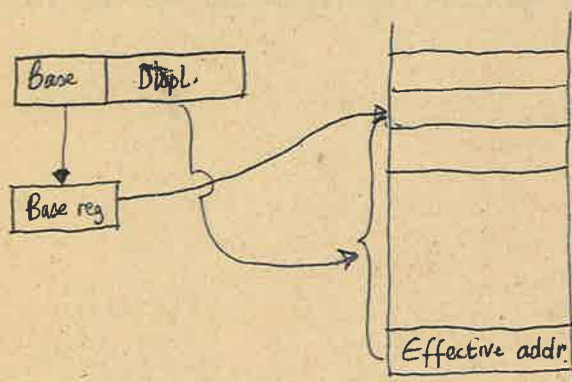
31101986

WATFIV Compiler

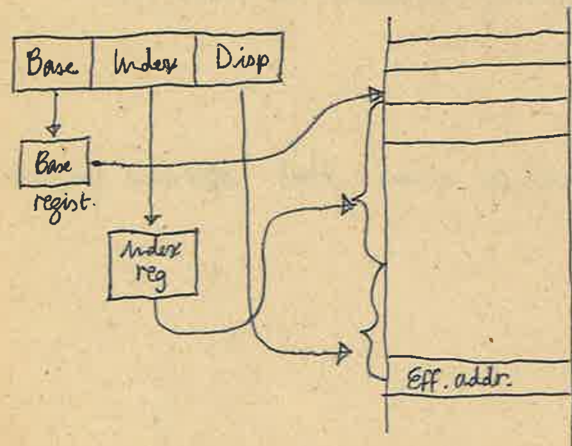
— A version of FORTRAN
— It contains a lot of debugging aids
— Extra Control structures (IF_THEN-ELSE, DO WHILE)

To run a WATFIV pgm:

i) Job card ⟶ //EF
ii) Batch mode //␢EXEC W



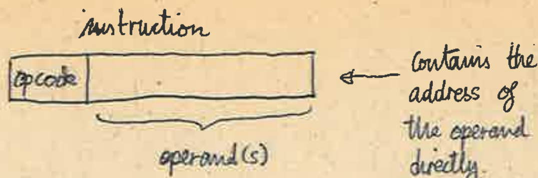
if you indicate R0 as base register or index, its content is ignored (i.e (R0) = 0 is taken).



For addressing, you have two choices;

i – absolute addressing. →

ii – relocatable addressing.



In each execution of an absolute program you should load it to a pre-specified location.

↓

the memory location to be reserved for the program is not pre-specified. The memory allocated to it may vary due to system condition. Computer (operating system) will do the memory allocation by looking at the various conditions, like memory usage.

Operating system: The program which controls the overall operation of the computer system.

Relocatable addressing is due to the base register. By changing the contents of the base register (Just before the execution of the program) you can load the program at any place in the memory.

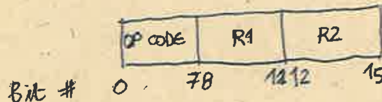
→ In this approach there is no pre-specified address to be occupied by the program.

Instruction Formats of IBM/370 (6 types)

By instruction formats we mean the structure of the instructions (operand organization)

1 – **RR type**

Both operand is register 2 bytes



Symbolic instruction: AR 5,7

(R5) ← (R7)+(R5)

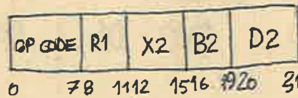
M/C instruction: 1A57

op-code for AR is 1A

2 – **RX type**:

First operand is in a register and second operand is in memory, during addressing both base and index registers are used.

4 bytes in length.



Explicit Symbolic instruction:

A 3, 36 (5,12) → explicit address specification.

A 3, NUMBER

implicit address specification, its conversion to explicit form is done by the ASSEMBLER

A 3, 36(5, 12)

↑ R1 ↑ D2 ↑ X2 ↑ B2

M/C instruction:

5A35C024

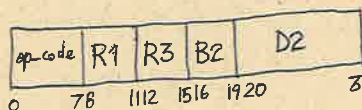
↑ opcode

3 – **RS type**

First operand in register, 2nd operand is in core (memory)

3rd operand in register.

4 bytes long.



Example:

Symbolic instruction STM 0,12,0(13)

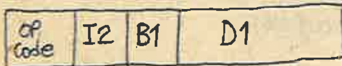
↑ R1 ↑ R3 ↑ D2 ↑ B2

M/C instruction → 900CD000

4 – SI type

(S = storage, I = immediate data (i.e. the data is available in the instruction))

1st operand in the core
2nd is in the instruction
4 bytes long.



Symbolic instruction:

MVI X'231'(12),C'b'

(X'231' = D1, 12 = B1; X'231' means '231' is hex; C means character)

M/C instruction:

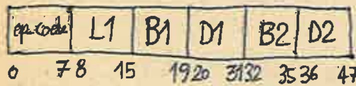
9240C231

EBCDIC form of 'b' → 40_{16}

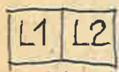
5 – SS type

Storage to storage type

1 and 2nd operands are in Core (memory)
6 bytes in length.



In some instructions, the length of both operands should be specified.

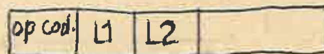


8 11 12 15

0–F 0–F

length → 1 16

In Some of the SS type instructions programmer can specify the length of both operands which are L1 and L2.



0 indicates length 1
↕
F indicates length 16

Example:

MVC D1(L,B1),D2(B2)

(L = L1)

MVC 0(15,12),16(1)

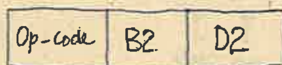
05111980

6 – S type

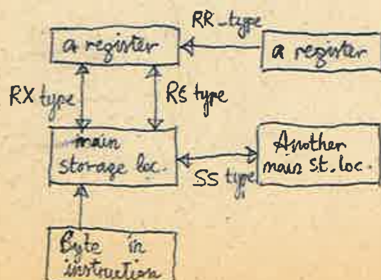
4 bytes in length

these instructions are special purpose, they cannot be used by the programmer. They are used by the operating System. This type of instructions are called as privileged instructions

OSR instruction!



Instructions:



Programming languages :

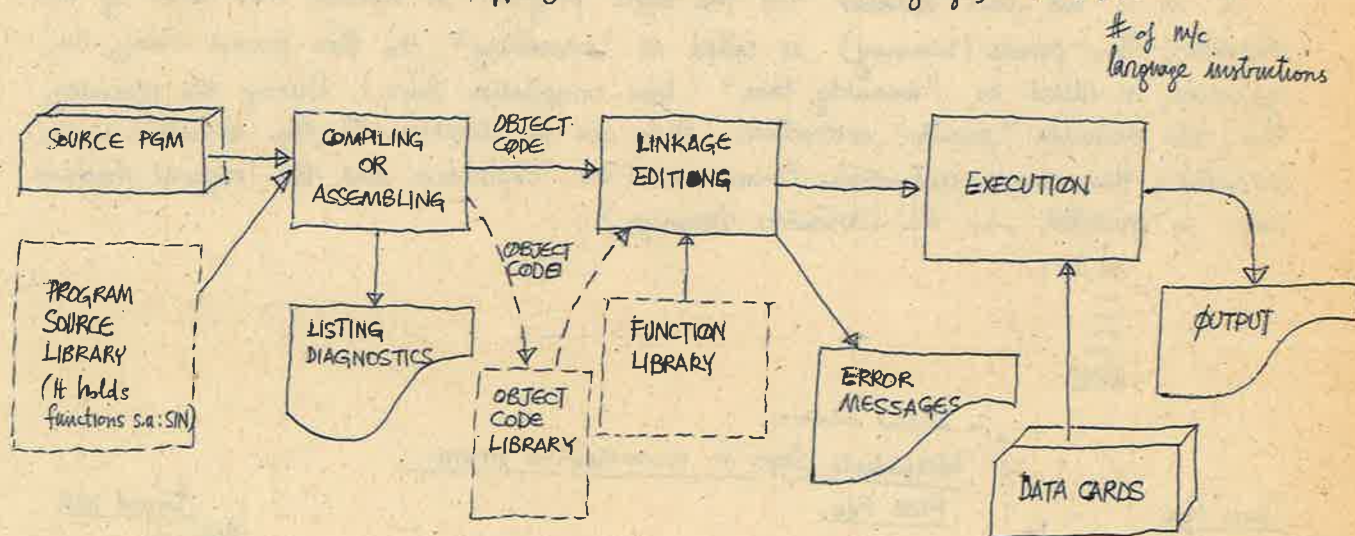
Machine language consists of 0's and 1's. To write an add register instruction for IBM 370 (eg, you want to do the following : (R5) ← (R5)+(R7)) ,

1A57

As an aid to the programmer we have symbolic programming languages.
Symbolic programming languages can be classified into two :
i – Assembler languages : the program written in assembly language , are nearly the symbolic form m/c lang.
ii – Higher level languages .

About assembler languages : In this Type of programs you will have 1 to 1 mapping . i.e : for each symbolic m/c instruction only one m/c instruction is produced.

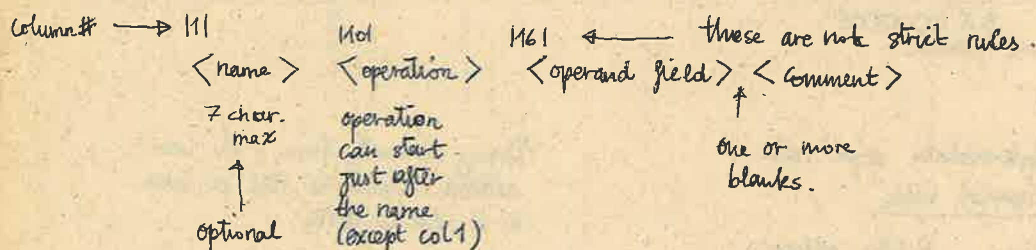
About higher level languages : Such as FORTRAN, PL/I, PASCAL
The mapping (translation into m/c language) 1 → N
\# of m/c language instructions



07111980

Format of the source program:

(Syntax Rules)



|1| L1 |10| AR |16| 5,7 can be written into this form : L1␣AR␣5,7

If a statement does not finish in a card, put a non_blank character into col. 72 then in the continuation card start at column 16. (col 1–15 are ignored) So in a card you can use columns 1-71 for programming purposes.

- Name field : Col 1–8
- Opcode (Mnemonic) of the instruction : col 10–14
- operand(s) : 16–71

After operands you may write comments, but there should be at least 1 blank in between.

You may write comment cards by entering an '*' to the first column.

The names (labels) can be at most 8 characters and the first character should be:

a letter : A-Z , $, @, #
↑ ↑ ↑
İ Ş Ö

The other characters of a name:

A-Z, İ, Ş, Ö, 0-9

In the operation field you can specify:

i- Constant and symbol definitions. (DC, DS) (pseudo instructions)

ii- Instructions to the assembler (pseudo instructions) __ END, START, USING

iii- Machine language instructions. (AR, MC)

iv- Macro instructions: for one macro instruction many m/c instructions can be generated and inserted into problem, they are also called as "open subroutine"

Macro Call.

Operation Principles of IBM assembler:

It is a two-pass assembler i.e the source program is scanned two times by the assembler. This process (scanning) is called as "assembling", the time passed during this operation is called as "assembly time" (like compilation time). During the assembly time, the assembler (pseudo) instructions - these are the directives to the assembler - are executed, the macro instructions (macro calls) are expanded and the required machine code is generated. In AL (assembler language):

```
START
 ═
 ═
END
```

Intermediate Steps in assembling a program

Source Pgm			Relative address (in decimal notation)	First Pass			Second pass Relative add.	
PROG	START							
	BALR	12,0	0	← two bytes in length 05C0			0	BALR 12,0
	USING	*,12	2	12: taken as base register; * indicates current value of location counter.				
	L	1,FIVE	2		{L R1,D2(X2,B2)} L 1,?(0,12)	4 bytes long.	2	L 1,18(0,12)
	A	1,FOUR	6		A 1,?(0,12)	4 bytes long.	6	A 1,14(0,12)
	ST	1,TEMP	10		ST 1,?(0,12)	4 bytes long.	10	ST 1,22(0,12)
	SR	2,2	14	SR 2,2		2 bytes long	14	SR 2,2
FOUR	DC	F'4'	16	4 X'0000004'				4
FIVE	DC	F'5'	20	5 X'0000005'				5
TEMP	DS	1F	24	one fullword				
	END		28					

Output of the first pass: 1- Intermediate object code.
2- Symbol table

Name	Relative address
PROG	0
FIVE	? → 16
FOUR	?
TEMP	?

During assembly time, the base address related to R12 is taken as 2, due to USING

Displacement = Relative address of the name (label) - the base address related with the base register.

— Continue from last lesson.

This program will be placed (loaded) in any place of memory. Assume that it is placed at location 12000_{10}.

Address	Contents of Memory	Symbolic M/C instruction
12000	05C0	BALR 12,0
12002	5810C012	L 1,18(0,12)
12006	5A10C00E	A 1,14(0,12)
12010	5010C016	ST 1,22(0,12)
12014	1B22	SR 2,2
12016	00000004	4
12020	00000005	5
12024	XXXXXXXX	PS F

L 1,18(0,12) ⟶ ≡ L 1,FIVE
During execution, the effective address for the second operand should point to the location of FIVE

BALR 12,0 instruction will load the address of the next instruction into R1 (in this case R12) and branches to the location pointed by the 2'nd operand R2 (in this case R0) But if reg2 is reg0 no branching will take place.

Effective address : (R0)+(R12)+18 ← displacement

↑ Since R0 is used as index reg., it has no effect; (R0) is assumed as 0

So ;

Effective address = 0+12002+18
= 12020

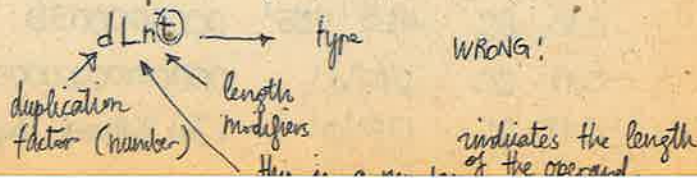
Note that the beginning address is loaded into the base register 12 by BALR 12,0 instruction. So the program can be loaded into different places in the memory in different executions of program. Due to the addressing characteristics (i.e base register and USING pseudo instruction) this will not effect the execution of program.
These kind of programs which can be placed in any memory location are called as 'relocatable'. The programs which should be placed into the same memory location are called as 'absolute'

Definition of constants in Assembler Language

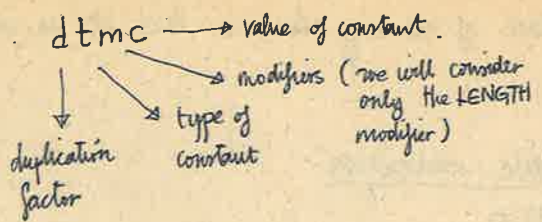
Name	Operation	Operands
optional constant name	DC	One or more operands separated by commas.

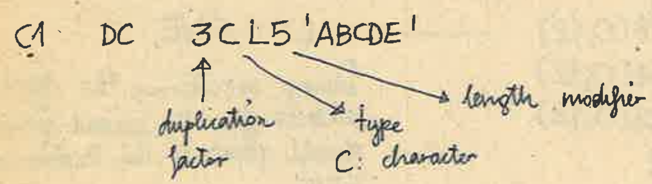
Each operand may have the following operand sub-fields,
not separated by commas or blanks:
i - duplication factor (optional)
ii - type of constant
iii - modifiers (optional)
iv - value of the constant

The operation field will be like:



In the operand field of DC (define constant) pseudo instruction we have the following sub-fields:





Examples : (For DC)

			In the memory:
C1	DC	C'125'	F1F2F5
C2	DC	CL2'4'	F440 (← this is due to padding, 40_{16} or b is padded for character type of constant)
C3	DC	2C'ABC' (↑ duplication factor)	C1C2C3C1C2C3
D1	DC	CL1'AB1'	C1
D2	DC	2CL1'AB1'	C1C1
C4	DC	X'ABCDE'	0ABCDE (↳ Padded hex digit.)
D2	DC	2XL2'FFB1C1'	B1C1B1C1
C5	DC	B'11001110'	CE
C6	DC	B'1111'	0F (← padded)
C7	DC	2BL2'00110011'	00330033 (due to duplication; 0 due to padding)

The length which can be specified in C, X, B type of data is 256 bytes.

			In the memory
C5	DC	H'155'	009B (binary value for 155)

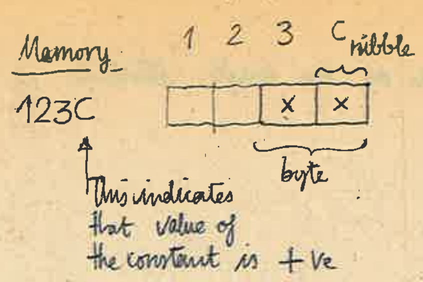
The range of numbers which can be specified as the constant value is between −32768 → 32767
In H type constants, there is halfword boundary alignment: Hw boundary means an address which can be divisible by 2

C6	DC	F'155'	0000009B	(fullword boundary alignment)
C7	DC	F'-155'	FFFFFF65	(" " ")
C8	DC	F'0'	00000000	(" " ")
C9	DC	FL3'155'	00009B	(no boundary alignment since length modifier prevents it)
C10	DC	HL5'155'	000000009B	(no boundary alignment)
C11	DC	2F'4'	0000000400000004	(fullword alignment)
C12	DC	0F'0'	(Just fullword alignment, no value is entered)	

Packed data

C13 DC P'123'

Memory 123C



ZEROS DC P'0000' 00000C
↑padded.

$$\# \text{ of packed bytes} = \frac{\# \text{ of digits specified} + 1}{2}$$

~~integer division~~

DC PL2'-73424' 424D → displays –ve
truncated.

Literals

A 3,ONE
⋮
ONE DC F'1'
≡
A 3,=F'1' (literal)
⋮
END

literal declarations have been done by assembler: 00000001 = F'1' After end statement

If the same literal is used more than once in a program, only one literal declaration will take place at the end of the program.

MVC LINE(15), EMPNO
EMPNO DC C'EMPLOYEE NUMBER'

MVC LINE(15), =C'EMPLOYEE NUMBER'

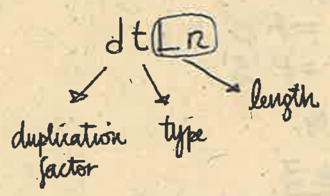
21111980

(R3) = FF000001 (a negative number)

STH 3,A A ← 0001
LH 4,A R4 ← 0000 0001
—
A DS H

DS → Define Storage

In the operand field we have the following subsections



storage to storage information movement:

MVC : Move character SS type 6 bytes long

MVC D1(L,B1),D2(B2)
MVC S1(L),S2 S1, S2 : names of variables
MVC S1, S2

MVC instruction operates as follows:

i) If the length is not specified it will be taken as the length attribute of S1 (This is valid for all SS type instructions)

ii) S2 is moved to S1

iii) Maximum length: 256.

iv) Movement of data is from left to right.

v) Data is moved one byte at a time.

MVI : Move immediate

SI type, 4 bytes in length.

```
MVI  D1(B1),I2
MVI  S1,S2
MVI  LINE,C'b'
```

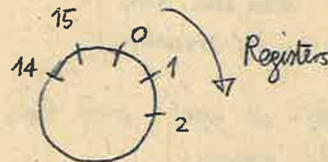
```
MVI  LINE,C'*'           } These instructions will fill all of the 'LINE'
MVC  LINE+1(131),LINE    } with *'s.
```

(LINE+1: displacement)

LM and STM instructions

LM R1, R3, D2(B2) load multiple, RS-type

STM R1, R3, D2(B2) " " , RS-type



LM 13,1,Q

R13 ← Q
R14 ← Q+4
R15 ← Q+8
R0 ← Q+12
R1 ← Q+16

Binary Arithmetic Instructions

Instruction	Full name	Type	Action
AR	Add register	RR	R1 ← (R1)+(R2)
SR	Subtract register	RR	R1 ← (R1)-(R2)
A (second operand in full word boundary)	Add	RX	R1 ← (R1)+(S2)
S (second operand in full word boundary)	Subtract	RX	R1 ← (R1)-(S2)
AH (second operand in h-w boundary)	Add halfword	RX	R1 ← (R1)+(S2) (0 ↔ 15 bits)
SH (second operand in h-w boundary)	Subtract halfword	RX	R1 ← (R1)-(S2) (0 ↔ 15)
MR	Multiply register	RR	R1‖R1+1 ← (R1+1)*(R2)
M (2nd operand in f.w. boundary)	Multiply	RX	R1‖R1+1 ← (R1+1)*(S2)
DR	Divide register	RR	R1+1 ← (R1‖R1+1)/(R2); R1 ← Remainder
D	Divide	RX	R1+1 ← (R1‖R1+1)/(S2); R1 ← Remainder

In MR, M, DR, D instructions, the register specified in the first operand (R1) must be an even numbered register.

Example 1

E = Y/(Q+3) Assume Y<0

```
L 9,Q
A 9,=F'3'
L 5,Y
L 4,=F'-1'
DR 4,9
ST 5,E
```

Example 2

Compute G as the remainder of P/3 (P>0)
In FORTRAN terms: G = MOD(P,3)

```
L 5,P
L 4,=F'0'
D 4,=F'3'
ST 4,G
```

Example 3

G = P/(Q+3) + Y/5 P>0, Y<0

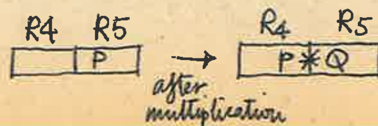
```
L 9,Q
A 9,=F'3'
L 5,P
L 4,=F'0'
DR 4,9
L 7,Y
L 6,=F'-1'
D 6,=F'5'
AR 5,7
ST 5,G
```

Example 4:

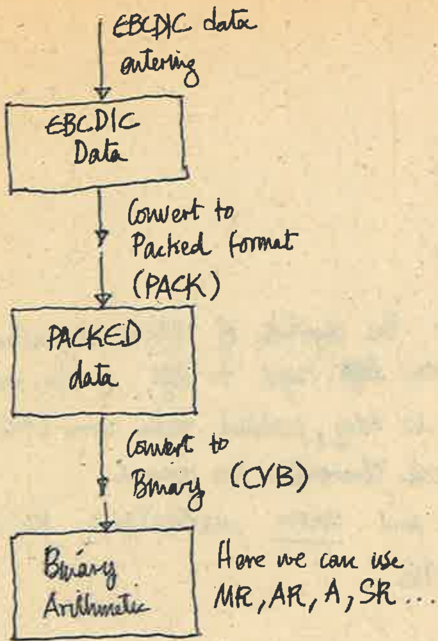
E = P*Q/Y

```
L 5,P
M 4,Q
D 4,Y
ST 5,E
```

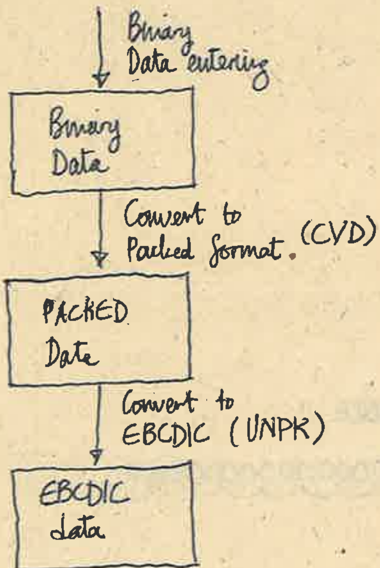
initially, contents of R4 will be ignored, it is assumed that odd numbered (in this case 5) contains the number to be multiplied



Steps to Convert EBCDIC Data into BINARY Format



Steps to Convert Binary Data into EBCDIC format



PACK instruction SS type 6 bytes long.

PACK D1(L1,B1), D2(L2,B2)
PACK S1(L1), S2(L2)
PACK S1, S2

← The length L1 and L2 will be taken as the length attributes of the variables S1, S2

This instruction converts the EBCDIC data of the 2nd operand into PACKED form and the result will appear in 1st operand.

Examples:

PACK A(3), INP(5)

INP → | F4 | F6 | F0 | F2 | F9 |

A → | 46 | 02 | 9F |

PACK E(4), INP+2(3)

+0 1 2 3 4

INP → | F4 | F6 | F0 | F2 | F9 |

Since displacement is defined as 2

E → | 00 | 00 | 02 | 9F |

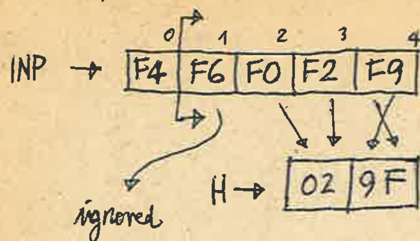
padding zeros supplied by PACK instruction

length of PACKED form = ⌈(length of EBCDIC form + 1)/2⌉

$\lceil 2.5 \rceil = 3$

$\lceil 5 \rceil = 5$

PACK H(2), INP+1(4)



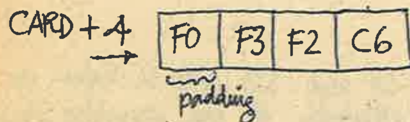
UNPACK instruction :

UNPK D1(L1,B1), D2(L2,B2)
UNPK S1(L1), S2(L2)
UNPK S1,S2
UNPK S1(L1), S2
UNPK S1, S2(L2)
SS type, 6 bytes

Unpack (UNPK) is the opposite of PACK instruction, execution is done from right to left, If the receiving field (1st operand) is long, padded with zeros (F0), if short, the unreceived characters are ignored.
Note that in PACK and UNPK instructions the lengths $0 < L1$ and $L2 \leq 16$

Examples :

UNPK CARD+4(4), PROD(2)



Convert to binary instruction

CVB R1, D2(X2,B2) RX type
Second operand, which is in double word boundary and 2 bytes (one double word) in length, which is a packed data converted into the binary form.
Recall that, last nibble of the rightmost byte of a packed data is stand for the sign of the number
If C,F +
D −

CVB 3, DOUBLE
(DOUBLE) = 000000000000047C
After CVB
(R3) = 0000002F

CVB 4, DOUBLE
(DOUBLE) = 000000000000025D → a −ve number.
after execution of CVB
(R4) = FFFFFFE7
DOUBLE defined as : DOUBLE DS D
(DS → define storage; D → double word boundary.)

Convert to Decimal :

CVD R1, D2(X2,B2) RX_Type
Binary data of (R1) is converted into PACKED decimal and inserted into 2nd operand
2nd operand should be in doubleword boundary, doubleword in length
CVD is exactly the reverse of CVB instruction

Example: Write a program which reads X and finds Y = MOD(X,10)

Read X from columns 6 through 10 of an input card, and produce the output as,

X = _ _ _ _ _

Y = MOD(X,10)

Y = _ _ _ _ _

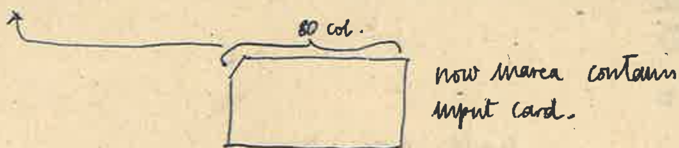
and assume that X is greater than zero

```
         PRINT NOGEN          ----> inhibit the expansion of macro instructions
PROG     START                (name of the program)
         STM   14,12,12(13)
                  -> R13 contains the beginning address of a 18-fullword
                     memory locations (supplied by OS)

         BALR  12,0     } Base register
         USING *,12     } is specified R12

         ST    13,SAVE13
         LA    13,SAVEAREA   <- puts the beginning address of SAVEAREA into R13

         look at Handout No1 !
*  READ AN INPUT CARD FROM CARD READER
         GET   INPT,INAREA   <- macro call.
```

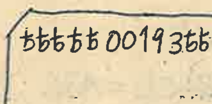


bbbbb0019355

INAREA : bbbbb0019355 --> (EBCDIC) code.

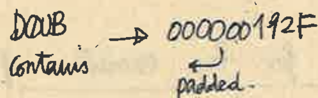
404040 ...

```
*  CHANGE EBCDIC DATA INTO PACKED FORM
         PACK  DOUB(8),INAREA+5(5)
```



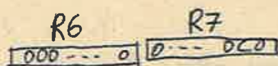
DOUB contains → 000000192F (padded.)

```
*  CONVERT TO BINARY
         CVB   7,DOUB
```



(R7) → 000000C0

```
         SR    6,6
```




```
         D     6,=F'10'
*  (R6) = REMAINDER (R7) = RESULT OF DIVISION (QUOTIENT)
         CVD   6,DOUB
*  CLEAN THE OUTPUT AREA
         MVI   OUT,C'b'
         MVC   OUT+1(131),OUT
```

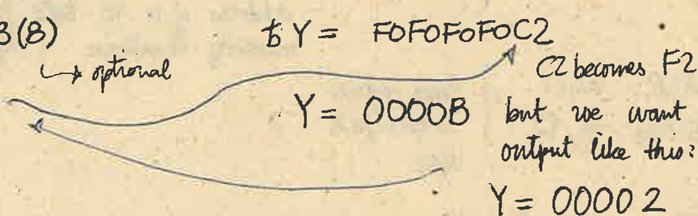
```
        MVC   OUT+1(2), =C'X='
  *  OUTPUT :  □X=□          (skip first byte)

        MVC   OUT+3(5), INAREA+5

  *  OUTPUT   ␢X = 00192
  *  OUTPUT   THIS LINE
Macro Call →  PUT   PRNT, OUT
        MVC   OUT+1(131),OUT        ← again we cleared the OUT

        MVC   OUT+1(11), =C'Y= MOD(X,10)'
Macro Call →  PUT   PRNT, OUT

        MVC   OUT+1(131),OUT        (CLEAN 'OUT')
        MVC   OUT+1(2),Y →   Y = 'Y='
        UNPK  OUT+3(5), DOUB(8)         ␢Y = FOFOFOFOC2
or immediate  OI    OUT+7, X'F0'   → optional      C2 becomes F2
        PUT   PRNT, OUT
```




```
  *  CLOSE I/O FILES
FINISH    CLOSE   (INPUT,, PRNT)
          L    13, SAVE13
          LM   14,12,12(13)
          BR   14       ← return or go to the address given by R14 (return address.)
OUT       DS   CL132
INAREA    DS   CL80
Y         DC   C'Y='
SAVE13    DS   F
SAVEAREA  DS   18F   →   DOUB  DS  D
  *  DECLARATIONS FOR I/O FILES
PRNT   SDCB  DDNAME = OUTDD, LRECL = 132, BLKSIZE = 132
INPT   SDCB  DDNAME = INDD, EODAD = FINISH
                                   (end of data go to finish)

       END        (short data control Block.)
```

Necessary Control cards (JCL) for the execution of this pgm.

```
//EF ... JOB ..       → compile (assembly)
//␢EXEC  ASMFCLG  ← go (execute)
              ↑  ↑ link run time macros
         F level assembler

// ASM.SYSIN  DD *   ← says pgm comes after this card.

      { Source pgm }

optional
→ /*                          it means printer
                                   ↓
// GO.OUTDD   DD     SYSOUT = A , DCB = RECFM = FA
// GO.INDD    DD  ← data definition.
                 *  → says that file (data cards) comes → immediately after this card.
```

Example Program :

```
REL
ADDR
0     PROG   START
0            STM    14,12,12(13)
4            BALR   12,0
6            USING  *,12     --> Pseudo instruction
6            SR     2,2
             A      2,FIVE
             ST     2,TEMP
             LM     14,12,12(13)
             BR     14
      FIVE   DC     F'5'
      TEMP   DS     F
             END
```

By using instruction, base address related with R12 will be taken as the current value of the location counter (due to * in USING)
Base address of R12 = 6
After USING, base register will be taken as R12

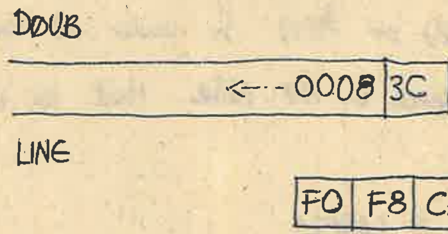
OI instruction

Convert the contents of R3 into EBCDIC code.
CVD, UNPK, OI

```
CVD    3,DOUB
UNPK   LINE(3),DOUB
PUT    PRNT,LINE        output 8C
OI     LINE+2,X'F0'
PUT    PRNT,LINE        output 83

DOUB   DS   D
LINE   DS   CL132
```

(R3) = 00000058



10.12.1980

Transfer of Control

PSW (Program Status Word)

Add cc.)

The condition code setting indicates whether the first operand is equal to, less or greater than the second operand and it also gives some information about an arithmetic operation.

Comparison Result	Condition Code
Equal	00
low	01
high	10

As it is told after some arithmetic operations cc will be set also

Arithmetic result	Condition code
zero balance	00
− ve	01
+ ve	10
overflow	11

The value of CC will be used in Branch instruction.

Branch on condition instructions :

```
BC    M1,D2(X2,B2)     RX-type
BCR   M1,R2            RR-type
```

If the CC value corresponds to the mask value go to the instruction whose address is given in the second operand.

Where M1 indicates a mask value for condition code setting $0 \leq M1 \leq 15$.

BCR is faster, since there is no efficient address calculation.

Mask field	Condition code	
8	0	=
4	1	<
2	2	>
1	3	Overflow.

```
        BC    4,MINUS
        ≡               → Pseudo instruction like FORTRAN "CONTINUE"
MINUS   EQU   *     } ≡ MINUS  AR  5,4
        AR    5,4   }
        BC    10,ZEROPOS
```

If CC is 0(8) or 2(2)   ( ) ← Mask.

go to ZEROPOS

```
BC   15,ALWAYS
```

If CC is 0(8) or 1(4) or 2(2) or 3(1) So under all conditions go to ALWAYS

If CC = 0(8) go to the address given in R7. Note that an address can be loaded into R7 by a LA instruction.

```
LA    7,LABEL1  } ≡ BC  8,LABEL1
BCR   8,7       }
```

```
BC    0,NOWHERE
```

Since mask cannot take a value like 0 above instruction does not cause any branching So it is like CONTINUE

```
BC    25,LABEL
```

This one causes a syntax error.

$0 \leq M \leq 15$

EXAMPLE : Output from the copy of the input data cards from line printer.

```
EXAMPLE   START
          ≡
          MVI    OUT,C'b'
          MVC    OUT+1(131),OUT
READ      EQU    *
INPT SDCB GET    INPT,CARD
EODAD BITTI MVC  OUT+10(80),CARD
          PUT    PRNT,OUT
          BC     15,READ  ————→  B READ
BITTI     EQU    *
          L      13,SAVE13
          LM
CARD      DS     CL80
OUT       DS     CL132
          ≡
          END    EXAMPLE
```

Extended Mnemonics:

BC, BCR

After comparison:

BC 2, <label>	BH <label>
BC 4,	BL
BC 8,	BE
BC 13,	BNH
BC 11,	BNL
BC 7,	BNE

After arithmetic:

BC 1,	BO
BC 2,	BP
BC 4,	BM
BC 8,	BZ
BC 13,	BNP
BC 11,	BNM
BC 7,	BNZ
BC 15	B (unconditional go)
BR 15,R2	BR R2
BC 0,	NOP
BCR 0,R2	NOPR R2

Binary Comparison

CR R1,R2 RR-type
C R1,D2(X2,B2) RX-type → Second operand should be in fullword boundary
CH R1,D2(X2,B2) RX-type

(D2(X2,B2) should be in halfword boundary.)

Condition code setting	Operand values
00	Both equal
01	first operand < 2nd opr.
10	first operand > 2nd opr.
11	not possible.

Compare logical instructions:

CLR R1,R2 RR-type
CL R1,D2(X2,B2) RX-type
(Second operand in fullword boundary)
CLI D1(B1), I2 SI-Type
CLC D1(L,B1), D2(B2) SS-type

Example: Add the contents of 10 integer numbers which are stored in the memory location named as ARRAY

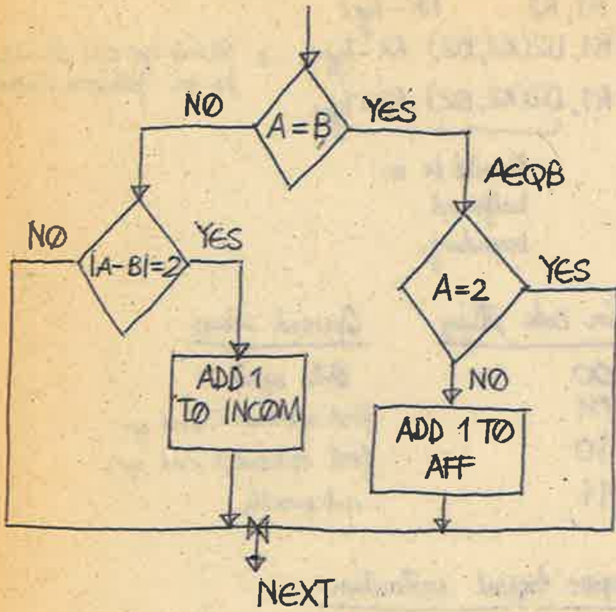
```
        ⋮
        SR   6,6        R6 WILL CONTAIN SUM
        SR   4,4        INDEX REGISTER
LOOP    EQU  *
        A    6,ARRAY(4)
        A    4,=H'4'
        C    4,=F'36'   ≡ CH 4,=H'36'
        BNH  LOOP
        CVD  6,DOUB
        UNPK OUT+10(5),DOUB+5(3)
        OI   OUT+14,X'F0'
        PUT  PRNT,OUT
        ⋮
ARRAY   DC   F'1,2,3,4,5,6,7,8,9,10' ——→ equivalent to: ARRAY DC F'1'
                                                             DC F'2'
```

Example:

CLR 1,2
(R1) = FFFFFFFF ↗ −1
(R2) = 00000001 → 1
In the logical comparison:
(R1) > (R2) so CC = 10

CLI OPERAND, X'C1'
OPERAND DC C'2'
↳ EBCDIC code for '2' is F2
So F2 > C1 again CC = 10

Example :




```
         L   4,A
         C   4,B        A=B ?
         BE  AEQB
         S   4,B
         LPR 4,4        (R4) = |A-B|
         C   4,=F'2'
         BNE NEXT
         L   5,INCOM
         A   5,=F'1'
         ST  5,INCOM
AEQB     EQU *
         C   4,=F'2'    (R4) = A
         BE  NEXT
         L   5,AFF
         A   5,=F'1'
         ST  5,AFF
NEXT     EQU *
```

logical instructions

type	AND	OR	EXOR	
RR	NR	OR	XR	
RX	N	O	X	(Second operand should be in fullword boundary)
SI	NI	OI	XI	
SS	NC	OC	XC	(Both operands are in memory)

```
NR   R1,R2
N    R1,D2(X2,B2)
NI   D1(B1),I2
NC   D1(L,B1),D2(B2)
          ↓
      0≤L≤256
```

Examples :

SR 2,2 ≡ XR 2,2

To clean a memory location :

```
MVI  OUT,C'b'
MVC  OUT+1(131),OUT
```
} instead of this : `XC OUT(132),OUT`

after execution, all bits of the memory location OUT will be set to zero

Since '00' is unprintable it appears as b at output.

looping instructions and indexing

Branch on Count	BCT	R1, D2(X2, B2)	RX
Branch on count register	BCTR	R1, R2	RR

↑ faster than BCT instruction since no calculation for eff. address of second operand is needed.

Operation of BCT, BCTR:

i) (R1) ← (R1) − 1

ii) If (R1) ≠ 0 branch to the address given in second operand.

else: next instruction is executed.

note that R1 is decremented before comparison by zero.

If R2 is R0 (in BCTR) no branching will take place. So

BCTR 6,0 ≡ S 6,=F'1'

↑ faster than S 6,=F'1'

Branch on Index low or Equal:

RS type BXLE R1,R3, D2(B2)

Branch on index high

RS type BXH R1,R3, D2(B2)

(D2(B2) = S2)

Operation of BXLE

(R1) ← (R1) + (R3)

if R3 is even (2,4,6....)

→ S2 if (R1) ≤ (R3) + 1

if R3 is odd numbered register

→ S2 if (R1) ≤ (R3)

Operation of BXH:

(R1) ← (R1) + (R3)

if R3 is even

→ S2 if (R1) > (R3+1)

if R3 is odd

→ S2 if (R1) > (R3)

Example:

Add 50 fullwords (in location A) into R5.

(R5) = A(1) + A(2) --- + A(50)

R11 : index register

R8 : increment register

In BXH:

R1 = reg 11

R3 = reg 8

R3+1 = reg 9

L 11, =A(ALAST) ≡ LA 11, ALAST

↑ indicates address.

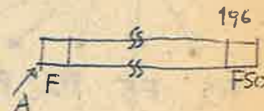
```
      L    9,=A(A)    ≡ LA 9,A
      S    9,=F'4'
      L    8,=F'-4'
      XR   5,5
LOOP  A    5,0(0,11)
      BXH  11,8,LOOP


A     DS   49F
ALAST DS   F
```

The same program with BXLE:

```
INDEX EQU  11
INCR  EQU  8
TESTR EQU  9
      SR   INDEX,INDEX
      L    INCR,=F'4'
      L    TEST,=F'196'
      SR   5,5
LOOP  EQU  *
      A    5,A(INDEX)   → A 5,A(11)

      BXLE INDEX,INCR,LOOP
      ↓
```



19.12.1980

Translate instruction :

TR D1(L,B1), D2(B2) SS type

↑ length of 1st operand (max 256)

Operation of the instruction :

do i = 1 to L

take the i-th byte of the first operand, add the numeric value of this byte into 2nd operand address, take the contents of the byte from this newly created address, move it to the i-th position of the first operand.

end.

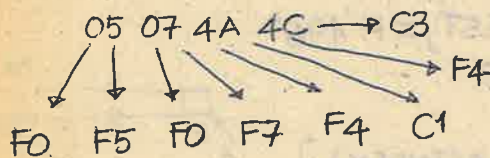
Example :

A → | 00 | 02 | 01 | 03 | 05 | 04 |
C7 D6 D5 E9 E4 D3 ← after TR instruction

B → | C7 | D5 | D6 | E9 | D3 | E4 |
G N O Z L U

∴ the new content of memory location A will be

| C7 | D6 | D5 | E9 | E4 | D3 |



Example : Write a program segment to give the hexadecimal dump of a program location called WORD with length of 4 bytes.

WORD : 05074A4C
5 5 ¢ <

To print the contents of WORD in hex. notation :

05 07 4A 4C → C3
F0 F5 F0 F7 F4 C1 F4

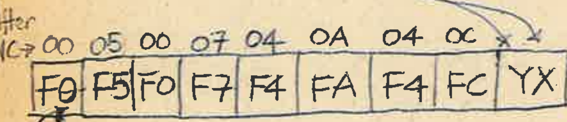
UNPK TEMP(9), WORD(5)

WORD → 05074A4C XY (X and Y are any hexadecimal digits)

After NC → 00 05 00 07 04 0A 04 0C

| F0 | F5 | F0 | F7 | F4 | FA | F4 | FC | YX |

TEMP



```
NC   TEMP(8),MASKF
TR   TEMP(8),TABLE
XC   OUT(132),OUT
MVC  OUT+5(8),TEMP
PUT  PRNT,OUT
```

After TR: F0 F5 F0 F7 F4 C1 F4 C3

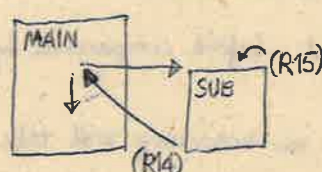
```
MASKF  DC  XL8'0F0F0F0F0F0F0F0F'
TABLE  DC  CL16'0123456789ABCDEF'
```

LINKAGE CONVENTIONS :

(Between a high level language and Assembly language)

The following general registers have special roles when performing program linkages :

R15 : Entry Point in the called pgm.
R14 : Return point in the calling pgm.



For linkage we should first initialize the contents of R15 with the beginning address of subprogram. And also R14 should contain the return address.

L 15, =V(literal ~~address~~ name of subroutine)

↓

here V indicates an external address constant.

L 15, =V(SUB1) will load the beginning address of SUB1 into R15

BALR 14,15

(R14) will contain address of next instruction, and execution will continue on the memory location pointed by R15.

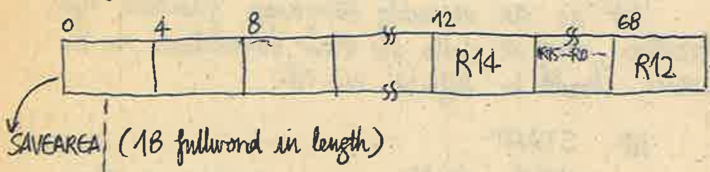
R13 : Address of a save area where the called program can save the general registers.
For saving, the called pgm. will execute
STM 14,12,12(13)

R0 : Returns the computed value in function references.

R1 : The address of the list of addresses corresponding to calling parameters.

According to the conventions established for IBM system 360/370 when every program and/or subroutine returns to the program that called it, the contents of general registers 12 through 14 must be the same as when the routine was entered.
For this purpose each calling program supplies a savearea to the called program. The lowest level subroutine may not supply a save area, by lowest level program which calls no other program.

Format of savearea:



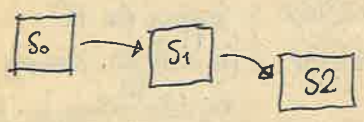
(18 fullword in length)

first fullword: ~~Contains the address~~ indicator used for only in PL/1 Programs.

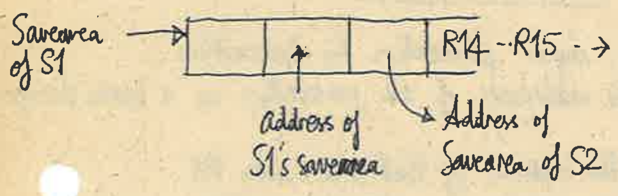
Second fullword: Contains the address of the calling routine's savearea (previous program)

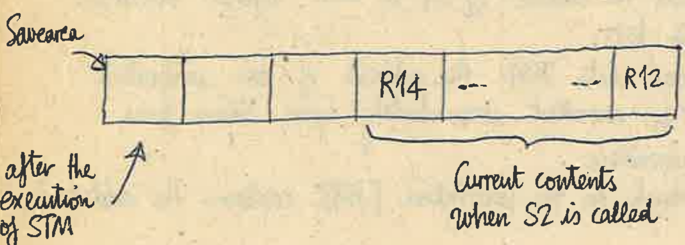
third fullword: Contains the address of the called subroutine's savearea. (Current Program)

Assume that S1 calls S2:



During the execution of S2, if we consider the contents of the savearea of S1 during that time.






```
STM  14,12,12(13)
```

Fortran statement like

CALL SUM(I,J,K)

will generate the following instructions.

```
          BAL   1,NEXT
AddressI  DC    A(I)
AddressJ  DC    A(J)
AddressK  DC    A(K)
NEXT      EQU   *
          L     15,=V(SUM)      external type address.
          BALR  14,15
```

After BAL 1,NEXT
(R1) points to the memory location which contains the address of I and branch to NEXT

L 15,=V(SUM)
will put the beginning address of SUM into R15

BALR 14,15
will put the return address into R14 and will branch to the address given in R15.

Example:

```
SUBROUTINE SUM(IRES,I1,I2)
IRES = I1+I2
RETURN
END
```

Assembler equivalent of SUM:

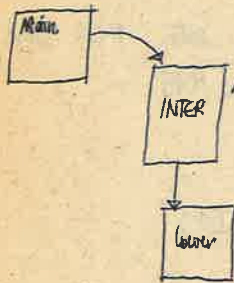
```
SUM    START
       USING  *,15
```

No need to load a value into R15 since it already contains the beginning address of SUM.

(R13): Beginning address of Savearea of the calling program.

```
      STM   14,12,12(13)
      L     2,0(1)
* (R2) = ADDRESS OF I
      L     3,4(1)
* (R3) = ADDRESS OF J
      L     3,0(3)
* (R3) = J
      L     4,8(1)      * (R4) = ADD OF K
      L     4,0(4)      * (R4) = K
      AR    3,4         * (R3) = J+K
      ST    3,0(2)
      LM    14,12,12(13)
      BR    14
      END
```

The main program written by the programmer is the Subprogram for operating system. So a savearea is supplied to the programmer in R13.
Let us write the linkage conventions should be obeyed by a program which calls other programs.



Consider this *intermediate* program

Since INTER calls another program, it cannot use R15 as the base register; It should supply a savearea in R13 to the lower level subprogram.

```
INTER   START
        STM    14,12,12(13)
        BALR   12,0
        USING  *,12
        ST     13,SAVE13
        LA     13,SAVEAREA
        ⋮
        L      13,SAVE13
        LM     14,12,12(13)
        BR     14
        ⋮               ──→  SAVEAREA   DS 18F
        END                  SAVE13     DS F
```

Example:

A program to compute I**2+K with I going from -10 to 10 in steps of 1.

```
      READ(5,1) K
1     FORMAT(I3)
      DO 10 I=1,21
      II=I-10
      N=NF(II,K)
      WRITE(6,2) II,N
2     FORMAT(3X,2I8)
10    CONTINUE
      STOP
      END

      FUNCTION NF(I,K)
      NF=I**2+K
      RETURN
      END
```

```
                CNOP  0,4
                BAL   1,NEXT
address of II   DC    A(II)
address of K    DC    A(K)
NEXT            EQU   *
                L     15,=V(NF)
                BALR  14,15
                ST    0,N
```

The instruction CNOP 0,4 is a no operation instruction but it sets the current value of the location counter to a fullword boundary.
So BAL 1,NEXT will be on fullword boundary.

Assembler equivalent of function NF:

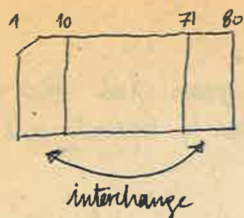
'NF' is an assembly language function type subroutine, and it calls no other subroutines, so no savearea should be defined in NF.

```
NF   START
     USING  *,15                1: address of address.
     STM    14,12,12(13)
     L      2,0(1)        **(R2) = ADDRESS 'II'
     L      5,0(2)        **(R5) = II
     MR     4,5   (R5)=II**2
     L      2,4(1)        **(R2) = ADDRESS 'K'
     L      0,0(2)        **(R0) = K
     AR     0,5           **(R0) = K+II**2
     LM     14,15,12(13)  } Skipping the R0,
     LM     1,12,24(13)   } (in order not to damage
                            the value stored in R0)
```

Subroutine linkage:

1. Prepare input parameters to subroutine.
2. Put the addresses of all parameters in a main storage area.
3. Load the address of that area into R1.
4. Load the address of an 18 Word register savearea into R13.
5. Load into R14 the address of the instruction to be executed immediately upon return from Subroutine
6. Branch to the subroutine (R15 contains the address)

Example: Write a program to do the following:



Main pgm: Assembler
Subprogram: FORTRAN

```
AMAIN    START
         STM 14,12,12(13)
         BALR 10,0
         USING *,10
         ST 13,SAVE13
         LA 13,SAVEAREA
         CALL READF,(ARG)
```

Macro → CALL; name of subroutine → READF

```
* In subroutine READF a data card is read and put
* into argument (ARG)
         MVC TEMP(10),ARG+70
         MVC ARG+70(10),ARG
         MVC ARG(10),TEMP

* Now print ARG by a FORTRAN subroutine called
* WRITEF,(ARG)  ←—— CALL WRITEF,(ARG)
         L 13,SAVE13
         LM 14,12,12(13)
         BR 14
ARG      DS 20F
SAVEAREA DS 18F
TEMP     DS CL10
SAVE13   DS F
         END AMAIN
```

```
      SUBROUTINE READF(A)
      INTEGER A(20)
      READ(5,1) A
    1 FORMAT (20A4)
      RETURN
      END

      SUBROUTINE WRITEF(B)
      INTEGER B(20)
      WRITE (6,1) B
    1 FORMAT (10X,20A4)
      RETURN
      END
```

Format of CALL macro:

i) If you don't pass any parameter to the called pgm.:
CALL <subroutine name>
Ex: CALL ALI

ii) If you pass parameters to the called pgm.:
CALL <subroutine name>,(<parameters>)
Ex: CALL VELI,(A)

Necessary JCL cards for this type of operations:

FORTRAN → ASSEMBLER communications

```
//EF... JOB ---
//b EXEC ASMFC
   { assembler  }    Repeat this for
   { subprogram }    all assembler
                     subroutines.

//b EXEC FORTGCLG,PARM.LKED='MAP'
   { FORTRAN                          optional
   { Main + subprograms.   (dumping of assembler subroutine)

//LKED.SYSIN DD DSN=&&OBJSET,      ← says continued
//bb DISP=(OLD,DELETE)
//GO.SYSIN DD *
   { Data cards (if any)
```

If you want to use an assembler program as a main program, write a dummy main FORTRAN pgm.

```
C     DUMMY MAIN
      CALL AMAIN
      STOP
      END
```

example:

Find smallest element of an array A.

```
C     FORTRAN MAIN
      INTEGER A(10)
      READ(5,10) A
   10 FORMAT(10I4)
      CALL SMALL(A,10,MIN)
C     "MIN" will contain the minimum element of A.
      WRITE(6,20) A,MIN
   20 FORMAT( ---
      STOP
      END

      SUBROUTINE SMALL (B,N,MIN)
      INTEGER B(N)
      MIN = B(1)
      L = N   (crossed out)
      DO 10 I = 2,N
   10 IF( B(I).LT.MIN) MIN = B(I)
      RETURN
      END
```

Assembler equivalent of SMALL:

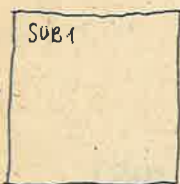
```
SMALL    START
*  R15 contains the beginning address of SMALL
         USING  *,15
         STM    14,12,12(13)
         L      2,0(1)     (R2): Beginning address of A
         L      3,4(1)
         L      3,0(3)     (R3) = N
*  Use Reg4 to contain the minimum element of A.
MINIMUM  EQU    4
         L      MINIMUM,0(2)
CONTINUE EQU    *
         S      3,=F'1'
         BZ     OKAY
         LA     2,4(2)     ≡ A  2,=F'4'
         C      MINIMUM,0(2)
         BNH    CONTINUE
         L      MINIMUM,0(2)
         B      CONTINUE
OKAY     EQU    *
         L      5,8(1)     (R5) = address of 'MIN'
         ST     MINIMUM,0(5)
         LM     14,12,12(13)
         BR     14
         END
```

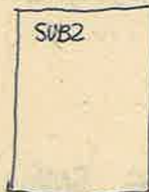
CALL SMALL (A,N,MIN)

equivalent assembly:

```
         L      15,=V(SMALL)
         CNOP   0,4
         BAL    1,NEXT
         DC     A(A)
         DC     A(N)
         DC     A(MIN)
NEXT     EQU    *
         BALR   14,15
```

There is another possibility to access the contents of an external routine.






```
SUB1     START
         ENTRY  SWITCH
         ≡
         ≡
         BR     14
SAVE13   DS     18F
SAVEAREA DS     18F
SWITCH   DS     CL1
         ≡
         END
```

ENTRY is a pseudo instruction

```
ENTRY    Var1,Var2 ---
```

Entry pseudo instruction means that the contents of the variable SWITCH can be accessed and changed by external routines.

Assume that subroutine SUB2 will acess to SWITCH

```
SUB2     START
         ≡
*  to change the contents of SWITCH
         L      3,=V(SWITCH)      (R3) = address of SWITCH.
*  'V' indicates that 'SWITCH' is an external var.
         MVI    0(3),X'01'     *(SWITCH) = X'01'
```

There is another pseudo instruction: EXTRN

```
         EXTRN  Var1,Var2 --
```

The variables which appear in the operand field are defined in external routines.

If we use equivalent of previous SUB2 by using EXTRN will be

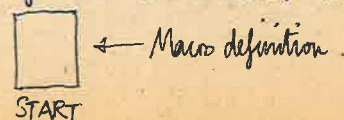
```
SUB2     START
         EXTRN  SWITCH
         ≡
         L      3,A(SWITCH)
         MVI    0(3),X'01'
```

MACROS:

Macro facility: Programmers have found that sets of codes occur repeatedly in their programs.
The programmer should not have to rewrite them each time. It is simple for him/her to give the code name and then require that this set of code is inserted wherever the name appears (replacing that name) The macro is the name which appears at opcode and which assembler replaces by a set of commands.

Macro types: System macros:
PUT, GET, SDCB declarations.
This type of macros are available in a disk file and the contents of that file can be accessed during assembly time.

Programmer macros:
Written by the programmer and it is put before the start 'START' instruction.



DATA STRUCTURES

data structures is designed to meet two basic requirements:
i- To represent the external information in a unique and unambiguous fashion.
ii- To facilitate efficient manipulation of the data by the computer.

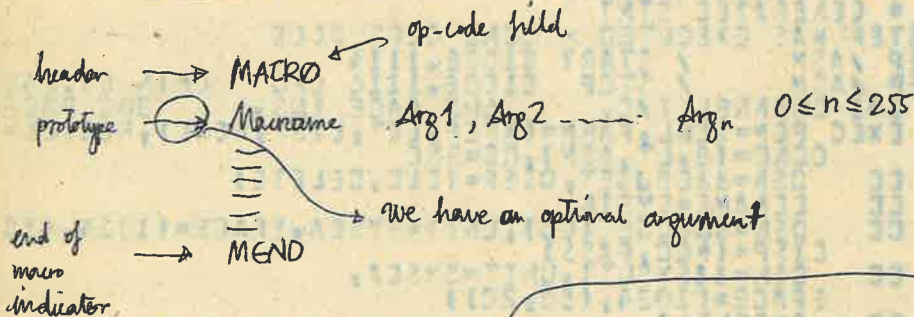
finding the address of last node:

```
   P = TOP = LAST
10 IF(P.EQ.0) RETURN
   LAST = P
   P = LINK(P)
   GOTO 10
   END
```

TOP: starting address.

EE439 A:

Macro definitions



Example:

```
      MACRO
&X    MOVEW   &A, &B
&X    L       5, &A
      ST      5, &B
      MEND
      START
      ...
      MOVEW   FROM,TO
+     L       5, FROM
+     ST      5, TO
      ...
      END
```

```
              MACRO
&ARGO         VARY    &COUNT, &ARG1, &ARG2, &ARG3
&ARGO         A       1, &ARG1
              AIF     (&COUNT b EQ b1).FINI
              A       2, &ARG2
              AIF     (&COUNT bEQ b2).FINI
              A       3, &ARG3
.FINI         ANOP  }  ≡ .FINI  MEND
              MEND  }
```

Macro Call 1:

```
   LOOP1   VARY   3, D1, D2, D3
+  LOOP1   A      1, D1
+          A      2, D2
+          A      3, D3
```

Macro Call 2:

```
           VARY   2, D3, D2
+          A      1, D3
+          A      2, D2
```

CJT

METU

Department of Electrical Engineering

EE 439-Digital Computers and

Symbolic Programming

Handout NO 1

In this write up three possibilities for executing an assembler language program is presented.

1. First possibility uses the SDCB's (Short Data Control Blocks):

```
//EE .... JØB, 'SURNAME, NAME', TIME=(m,s)
//EXEC  ASMFCLG
//ASM.SYSIN DD *
            PRINT  NØGEN
      PRØG  START
            STM    14,12,12(13)
            BALR   12,0
            USING  *,12
            ST     13,SAVE13
            LA     13,SAVEAREA
            OPEN   (INPT,(INPUT),PRNT,(OUTPUT))
      INSERT YOUR PROGRAM SEGMENT HERE.
            CLOSE  (INPT,,PRNT)
            L      13,SAVE13
            LM     14,12,12(13)
            BR     14
     PRINT  SDCB   DDNAME=OUTDD,LRECL=132,BLKSIZE=132
     INPT   SDCB   DDNAME=INDD,EØDAD=FINISH
     SAVEAREA DS   18F
     SAVE13 DS     F
     CARD   DS     CL80
     LINE   DS     CL132
```

Handwritten annotations: "comment" (arrow to OPEN line); "* OPEN INPUT OUTPUT FILES"; "← a macro call." (beside OPEN); "↑ we can use any name" (under INPT); "EODAD" handwritten over struck-out "EOBDAD".

```
         Other DS and DC declarations
                  END      PRØG
/*
//GØ.ØUTDD        DD SYSØUT=A, DCB=RECFM=FA
//GØ.INDD  DD *
         Data cards (if any).
```

In order to read a card from card creater into the area CARD code

GET INPT, CARD

In order to print a line by line printer use the area ØUT (132 bytes).

PUT PRNT, LINE

EØDAD=FINISH is like FØRTRAN END option, ie,

READ(5,10,END=100)

When data cards finished go to statement number 100 (FINISH in our case).

In assembler program if you do not want to use end option, do not code ",EØDAD=FINISH". If you use this option, somewhere in your program there has to be a label named FINISH, or any name you like, i.e., you can use other labels instead of finish, so it can be EØDAD=BITTI or any other name.

In printing, the first character of the area which is outputted is used for carriage control (in this example the first ~~position~~ byte of LINE). The common codes and their actions are as follows:

blank	normal single spacing
0	double space
-	triple space
+	suppress spacing
1	ship to the top of the next page.

2. Second possibility for executing an assembly language program:

```
// EF ... JØB ...
// EXEC ASM311
PRØG BEGIN
     ENDFILE FINISH

   INSERT YOUR PROGRAM SEGMENT HERE
     EØJ
CARD DS   CL80
LINE DS   CL132

          Other DS and DC declarations
     END  PRØG
//GØ.SYSIN  DD  *
        Data cards if any
```

In order to read a card from card reader into the area CARD code:

```
INPUT CARD
```

in order to print a line by line printer from the area ØUT(132 bytes) code.

```
ØUTPUT LINE
```

ENDFILE label card is again optional, it is like EØDAD=FINISH. In this usage if there is no more card to be read program directly goes to EØJ command (ie, stop the execution).

For carriage control the arguments given above are also valid.

Note that BEGIN macro uses the R13 as the base register so do not use it in calculations.

If you want to see the expension of BEGIN macro you must code it as follows:

```
label       BEGIN  P=X
```

You may specify one more base register by BEGIN macro eg, if you want to specify R12 as the second base register code it as:

```
label       BEGIN   B=12
```

3. Third possibility for executing an assembly program:

```
// EF ... JØB ...
// EXEC    A
PRØG       BEGIN

     INSERT YØUR PRØGRAM SEGMENT HERE

           EØJ
CARD       DS       CL80
LINE       DS       CL132

     Other DS and DC declarations.
           END PRØG
     Data cards if any
```

BASE REGISTER : 13

"// EXEC A" card invokes an incore assembler, due to this reason it provides more debugging aids, note that WATFIV is an incore ~~computer~~ compiler also, and a lot of debugging facilities of WATFIV again comes from this incore feature.

Arguments given for BEGIN ~~is~~ are also valid for this case, but there is no end of file option, if end of data cards is reached program directly goes to EØJ , so execution will stop. Input and output is done as it is presented in second possibility.

Middle East Technical University

Electrical Engineering Department

EE-439

Homework-4

Date Assigned : Dec. 12th, 1980

Date Due : Dec. 24th, 1980

Code and run an IBM/370 Assembly Language program to count the bytes which contains X'00',, X'01', X'02,...,X'09'.

A.Ş

Arrange your output as follows:

BYTE	FREQUENCY
00	.
01	.
.	.
.	.
.	.
09	.

Note that to see the contents of the object code generated for macro calls you should remove the 'PRINT NOGEN' card. But during the initial phases of the program development you may use it to prevent paper wastage.

At the end, you will see that the number of (say) X'00' bytes which appears on the program listing is different than the one which is found by your program. This is due to the address constant, like

→ address constant

```
DC    AL3(PRNT)
```

This type of constants are modified by the loader according to the memory location in which your program has been loaded.

The address constants like AL3(PRNT) will be modified by the loader when the program is loaded into the memory.
Assume that your program is loaded at 1A 0000
So the real address of label PRNT will be 1A0100 (0100: relative location)

So during execution time the memory location corresponding to DC AL3(PRNT) will contain 1A0100
Notice that first two hexadecimal digits are not 00 that is why $a \neq b$

a: # of 00's in program listing
b: # of 00's in the program during execution

V

```
PROG    START
        STM    14,12,12(13)
        BALR   12,0
        USING  *,12
        ST     13,SAVE13
        LA     13,SAVEAREA
* BEGIN
        LA     10,PROG
        LA     11,ENDLAB
        SR     3,3      (R3) WILL BE '00' '01' ... '09'
        SR     4,4      R4 WILL BE USED FOR SUMMATION
LOOP    EQU    *        ----------> TEMP  DC  H'0'
        MVC    TEMP+1(1),0(10)        0000
                                      0090
        CH     3,TEMP
        BNE    NEXT
        A      4,=F'1'
NEXT    EQU    *
        AH     10,=H'1'   Movement to the next Memory location
        CR     10,11
        BNH    LOOP

        [ OUTPUT ]

TEMP    DC     H'0'
ENDLAB  EQU    *
        END
```