INTCODE

documentation:

henrik andersen
kurt jensen
børge s. kirk

CONTENTS

INTCODE

STORE AND REGISTERS

The intcode-machine is designed as a tool to help implementing BCPL on a new machine.

The intcode-machine has a store consisting of equal sized locations addressed by consecutive integers.
When emulated at a 16 bit machine as RIKKE it is natural to use two different instructionformats. Long format uses two consecutive words, while short format only uses one.
The choice between these formats is automatically made by the assembler.

The intcode-machine has 6 different 16 bit registers:

A	accumulator	This register can be loaded from store.
B	auxiliary accumulator	When A is loaded the old content of A is moved to B. All operations involving two operands are performed with these taken from B and A (result in A).
C	program counter	Points to next instruction to be executed.
D	address	Keeps the effective (resulting) address.
P	run-time stack	Points to the bottom element of run-time stack. Is used to reference elements on this stack (local variables).
G	global	Points to first element in globalvector. Is used to reference elements in global-vector.

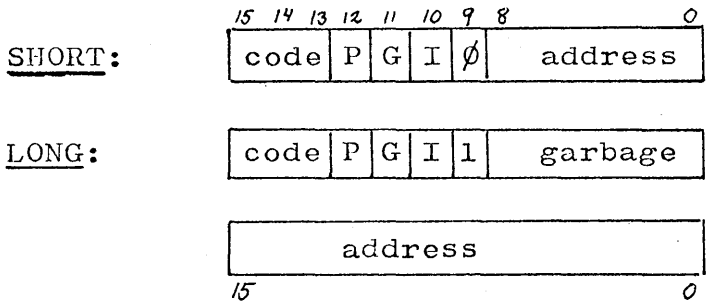
INSTRUCTION FORMATS

Each instruction consits of six fields as follows:

instruction code:	3 bits giving 8 possibilities
address field:	9 or 16 bit (depending of long or short format). In both cases the field is interpreted as a nonnegative integer.
P-bit	iff set P-register is added to the address.
G-bit	iff set G-register is added to the address.
I-bit	iff set the address, D, should be replaced by loc(D) (indirect addressing). This is done after possible adding of P and G.
L-bit	iff set long instructionformat is used.

The effective address is calculated in 4 steps:

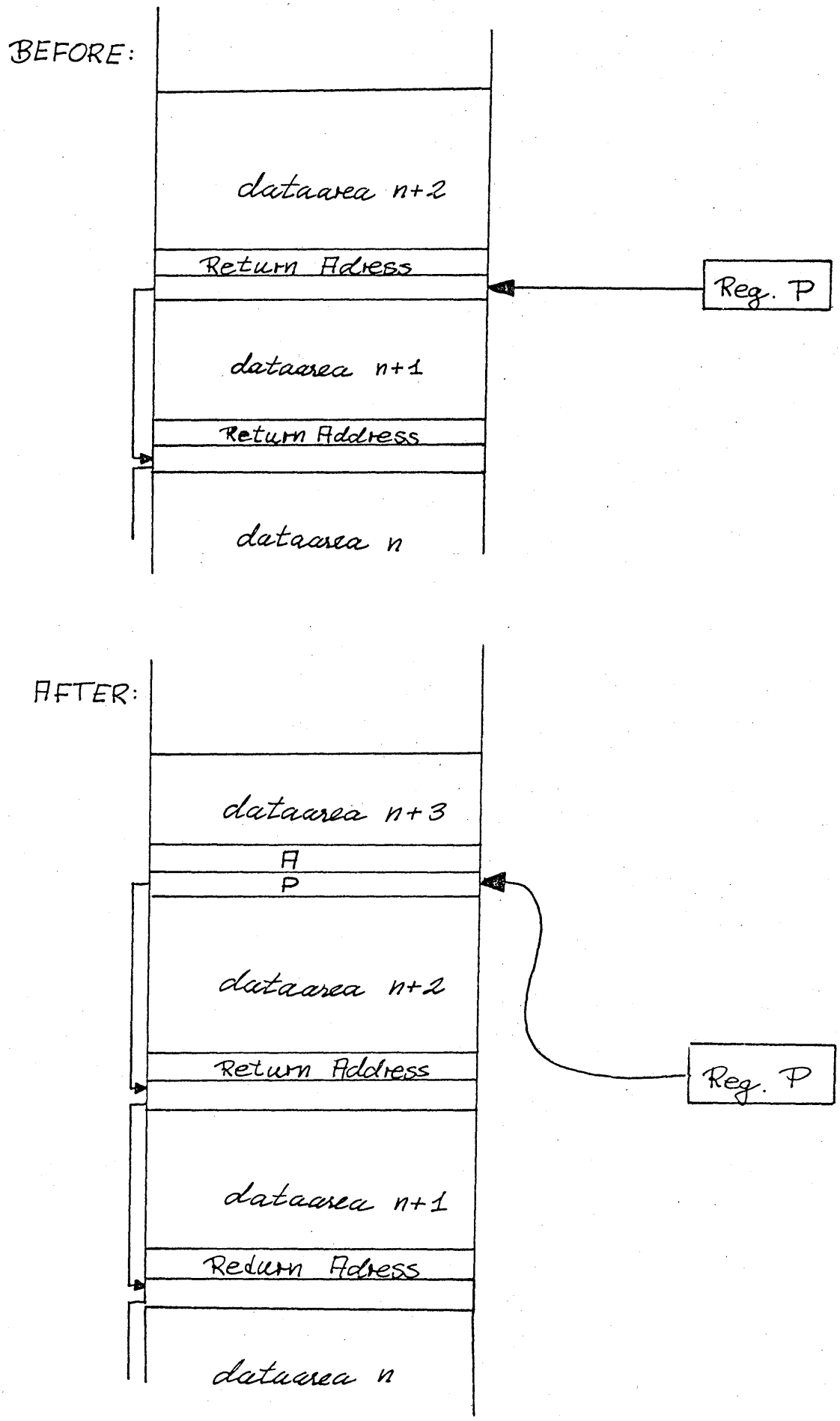
1. D:= addressfield (depending on L-bit)
2. possible adding of P-register
3. possible adding of G-register
4. possible indirectness



INSTRUCTIONCODES

code	mnemonic	description	comments
000	Load	B:=A; A:=D	This is not a "normal" load-instruction. It loads the effective address and not the content of this address.
001	Store	loc(D):= A	
010	Add	A := A + D	
011	Jump	C := D	Unconditional jump to absolute address D.
100	jump if True	if A=0 then C:=D	Conditional jump to absolute address D.
101	jump if False	if A≠0 then C:=D	
110	K	loc(P+D):= P loc(P+D+1):= C P:= P+D C:= A	Recursive function call. The current stack frame is specified by D and the entry point is given in A. The first two cells of the new stack frame are set to hold return link information.
111	eXecute		Allows auxiliary operations to be executed. The operation is specified by D modulu 32 (last five bits).

Figure showing the effect of a K-instruction



EXECUTEOPERATIONS

no.	description	comments
X1	A:= loc(A)	
X2	A:= -A	
X3	A:= not A	
X4	C:= loc(P+1) P:= loc(P)	Return from current routine or function. Result of function is left in A.
X5	A:= B * A	
X6	A:= B / A	Integer division
X7	A:= B rem A	Rest after integer division.
X8	A:= B + A	
X9	A:= B - A	
X10	A:= B = A	These operations yields a boolean result (true or false)
X11	A:= B ≠ A	
X12	A:= B < A	
X13	A:= B ⩾ A	
X14	A:= B > A	
X15	A:= B ⩽ A	
X16	A:= B lshift A	B is shifted A times logical left
X17	A:= B rshift A	B is shifted A times logical right "B lshift A" is equivalent to "B rshift -A"
X18	A:= B ∧ A	
X19	A:= B ∨ A	
X20	A:= B ≢ A	
X21	A:= B ≡ A	
X22	finish	

X23 switch

This operation switches between a set of labels while it tests A against a set of conditions.

The data following X23 is used in the following way.

```
X23
n
label.default
condition.1
label.1
condition.2
label.2
   :
condition.n
label.n
```

n is a nonnegative integer.
Label.default, label.1 .... label.n are labelvalues.
Condition.1 ..... condition.n are integers.

The above peace of code is equivalent to:

```
if A = condition.1 goto label.1
if A = condition.n goto label.n
     :
if A = condition.n goto label.n
goto label.default
```

X24 selectinput — selects inputdevice according to A

X25 input — input data to A from selected inputdevice

X26 selectoutput — selects outputdevice according to A

X27 output — output data from A to selected outputdevice

ASSEMBLER

SYNTAX - INFORMAL DESCRIPTION

Input to the assembler is a program written in intcode (symbolic intcode using mnemonics).

A program is a sequence of one ore more segments each consisting of a sequence of intcode-instructions terminated by a special Z-instruction.

Instructions can be of 6 different kinds:

normal-instruc	:	executable instructions such as load, store etc.
D-instruction	:	pseudoinstruction places data in dataarea
C-instruction	:	pseudoinstruction places character values in dataarea (packed two and two)
G-instruction	:	pseudoinstruction initializes the globalvector (only with labelvalues)
L-instruction	:	pseudoinstruction - list and nolist
Z-instruction	:	pseudoinstruction terminates a segment

A normal-instruction consits of

a) label
b) instructioncode
c) IPG-bits
d) address

where IPG-bits are mnemonics for

I - indirect addressing
P - add P-register to address
G - add G-register to address

Address can be either an absolut address (integer) or a label.

b) and d) are always present while a) and c) can be omited.

D-instruction: places datavalue in memorycell in dataarea.

When an integer is prefixed by D this integer is placed in the next cell in dataarea.

When a labelnumber is prefixed by DL the value of this labelnumber is placed in the next cell in dataarea.

C-instruction: is used to pack charactervalues in dataarea. Each character value is prefixed by a C.

The charactervalues are packed left to right, two in each cell.

If the left half of a cell has been filled and the next instruction is not a C-instruction without a labeldeclaration the right half will be padded with zeros.

G-instruction: is used to initialize element of globalvector with the value of a labelnumber.

The format is : G <gloablnumber> L <labelnumber>

L-instruction: is either an Y or a N.

Y (yes) starts listing of codetext on outputfile
N (no) stops listing of codetext on outputfile

Z-instruction: terminates a segment and cannot be used anywhere else.

SYNTAX - FORMAL DESCRIPTION

The syntax is now described in BNF.

Underlined symbols are terminals.
{n,m} menas an integer in the closed interval [n,m]

Everywhere in the sourcetext / can be inserted - then this character and the following until the next lineshift are skipped.
This can be used as comment- or continuation facility.

Each label can only be declared once.
Each referenced label must be declared.

```
<program>        ::= <segment>+
<segment>        ::= <instruction>*<Z-instruc>

<instruction>    ::= <normal-instruc> |
                     <D-instruction> |
                     <C-instruction> |
                     <G-instruction> |
                     <L-instruction>

<normal-instruc> ::= <labelpart><instruccode> <IPG-bit><address>
<D-instruction>  ::= <labelpart> D <data>
<C-instruction>  ::= <labelpart> C <charvalue>
<G-instruction>  ::= <skip> G<global> L<label>
<L-instruction>  ::= <skip>(Y|N)
<Z-instruction>  ::= <skip> Z  <skip>

<labelpart>      ::=  <skip> (<label> <delim>)*
<skip>           ::= (*s|*n| $)*
<delim>          ::= (*s|*n| $)+
<label>          ::= {1,500}

<IPG-bit>        ::= (I|P|G)*
<address>        ::= {0,  64K-1} | L <label>

<instruccode>    ::= (L|S|A|J|T|F|K|X)
```

<data> ::= {-32K , 32K-1} | L <label>
<charvalue> ::= {0,255}
<global> ::= {0,511}

STRUCTURE OF ASSEMBLER

The program is devided into 3 parts

asshead keeps all global and manifest declarations

assproc keeps all procedures

assmain keeps the main program

assproc and assmain uses asshead by a get-directive

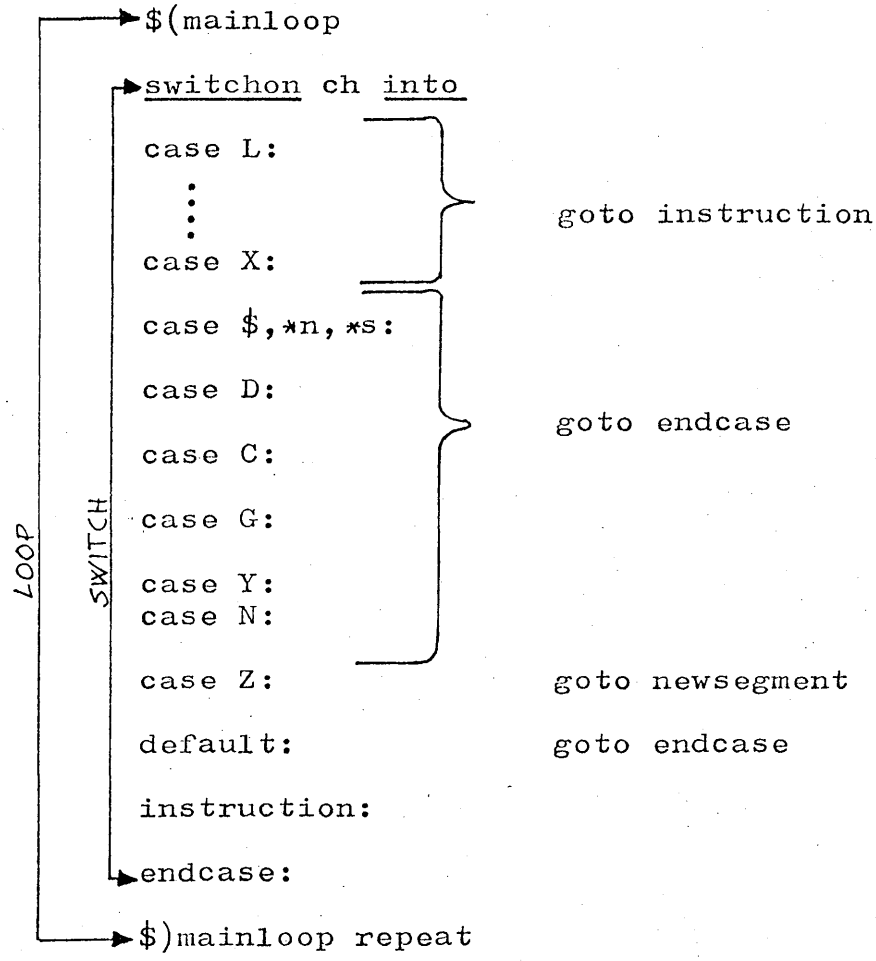
For information about the procedures in assproc please see the comments in the BCPL-listing.

assmain has this structure:

declarations and initializations

read()

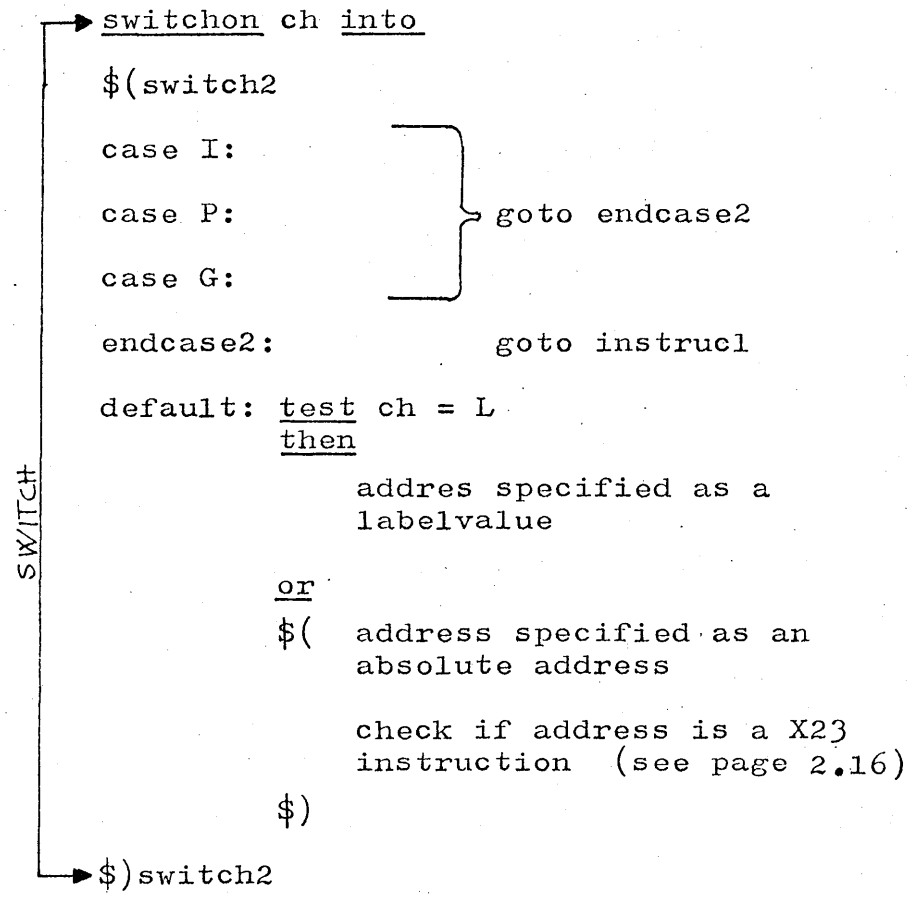
newsegment: more initialization - presentation



If an L,S, ..., X is recognized it is a normal-instruction. After this letter has been recognized these instructions are handled together in "instruction".

"instruction" has this structure:

```
instruction:    listing on output
instrucl:       read()
```



case S,*n,*s: handles insignificant characters. These are skipped.

case D: handles D-instructions. The structure is:

```
case D:  read()
         switchon ch into
         $(switch1
         case L:  data is specified as a labelvalue
                  goto endcase  (see bottom page 2.5)
         default: data specified as an
                  integer value
                  goto endcase
         $)switch1
```

SWITCH

case C: handles C-instructions. The structure is:

```
case C:  read()
         collect first charvalue
         collect second charvalue (if any)
         pack these together and put them in
         dataarea
         goto endcase   (see bottom of page 2.5)
```

case Y: and case N: handles L-instructions and is very simple.

case G: handles G-instructions. The structure is:

```
case G: read()
        collect globalnumber
        check if "L" is present
        collect labelnumber
        call insertref to update pointerchain
        goto endcase  (see bottom of page  2.5)
```

case Z: handles Z-instructions. This signals the end of a segment. The structure is:

```
case Z:  check if any referenced label still is undeclared

         unless error do  dump segmentblock (see page 2.9)

         test "more segments"

         then  goto newsegment

         or  $(  mark 1-0
                 mark 1-0
                 closeall()
                 finish

             $)
```

default: handles labeldeclarations (or illegal sourcetext characters). The structure is:

```
default:   while ch = tal
           $(
              collect labelnumber (declaration)
              update dectabarea  (see page 2.13)
           $)

           switchon ch into

           $(sw

           case D:
           case C:          } declaration-address in dataarea

           case L:
             :              } declaration-address in codearea
             .
           case X:

           the chained cells in dectabarea are all set to
           point to the  declaration address     (see page 2.13)

           goto endcase  (see bottom of page 2.5)

           case G, Z, Y, N:  illegal labeldeclaration
                             goto endcase

           default:  illegal sourcetext character
                     goto endcase

           $)sw
```

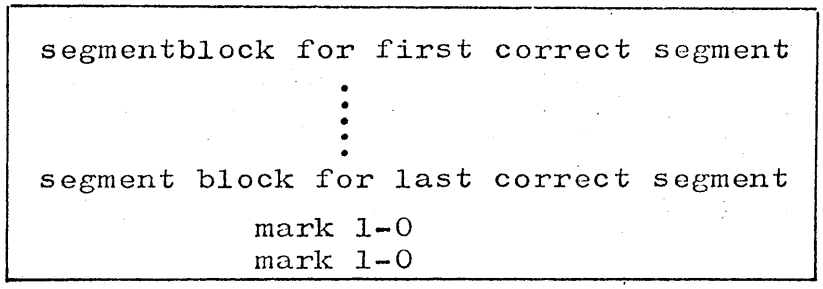
SWITCH

OUTPUT OF ASSEMBLER

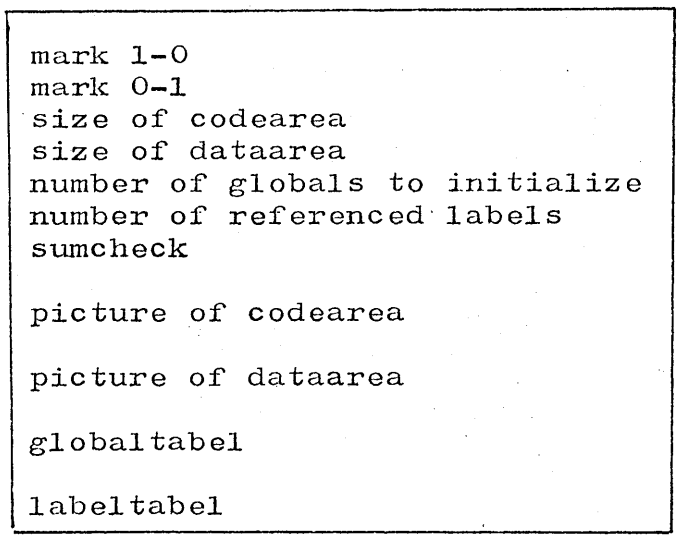
The assembler translates each segment as a unit. Listing of sourcetext and errormessages will be on file output (according to the pseudoinstructions Y and N).

If one or more segments is correct the assembler will deliver a file which will be accepted as input to the intcode-loader.

This file will have the following format.



where segmentblock is



mark 1-0 is 1111111111111100
mark 0-1 is 0000000000000011

Instructions are separated in two areas:

codearea keeps all normal instructions

dataarea keeps D-instructions and C-instructions

all other instructions are pseudoinstructions which do not need any space in mainstore.

Sumcheck is the sum of all words in the following four tabels (codearea,dataarea,globaltabel and labeltabel).

Codearea and dataarea is simply a picture of these two areas. For problems about labelreferences and labeldeclarations please see page 2.13.

Globaltabel keeps information about which elements of the globalvector should be initialized.

The format of globaltabel is:

1. globalno
1. value
2. globalno
2. value
:
n. globalno
n. value

where n = "number of globals to initialize".

For each i between 1 and n i. globalno keeps the number of the global which should be intialized; and i. value gives a value which is put into the globalvector. This value is part of a pointerchain between labelreferences which later on by the loader will be altered to an absolut mainstore address.

Labeltabel keeps information about labeldeclarations and label-references.

The format is:

1. labeldec
1. labelref
2. labeldec
2. labelref
⋮
n. labeldec
n. labelref

n = "number of referenced labels".

For each referenced label there are two corresponding words in the tabel (i. labeldec and i. labelref) from which all nescessary information about use of this label can be extracted.

i.labeldec gives the place (in dataarea or codearea) where this label is declared.

i.labelref is head of a one-way list which gives the places (in dataarea, codearea or globalvector) where this label is referenced. The list is terminated by a special mark (1111111111111101).

The number of the label is insignificant and is not given in the output of the assembler.

WORKINGTABLES

The assembler has 7 tables to keep information about a segment.

Codearea keeps all normal-instructions. Cpointer points to the first empty word.

Dataarea keeps all D-instructions and C-instructions. Cpointer points to the first empty word.

Globarea keeps information about all initialized globals.
The index corresponds to the globalnumber.
The value is part of a list connecting all places (in codearea, dataarea and globarea) where this particular label is referenced.

Dectabarea and Reftabarea keeps information about declaration and referencing of labels. The index corresponds to the labelnumber. For detailed information see page .

Decarea and Refarea do exactly the same as Dectabarea and Reftabarea, but the former (dectabarea and reftabarea) is handling userdefined labels, while decarea and refarea handles assembler generated labels (see p 2.16).Lpointer points to the first free labelname in decarea and refarea.

When a correct segment is fineshed the following is dumped:

codearea	as	"picture of codearea"
dataarea	as	"picture of dataarea"
cpointer	as	"size of codearea"
dpointer	as	"size of dataarea"

Globarea is scanned and for each initialized global its number and its value is dumped as part of "globaltabel".

The number of such globals is dumped as "number of globals to initialize".

Dectabarea and reftabarea is scanned and for each referenced label, i , dectabarea.i and reftabarea.i is dumped as part of labeltabel.

The used part of decarea and refarea are dumped as part of labeltabel in the same manner as for dectabarea and reftabarea.

The number of referenced labels in reftabarea and refarea is dumped as "number of referenced labels".

LABELDECLARATIONS AND LABELREFERNECES

When a labeldeclaration is met it is nescessary to know whether it is preceding an instruction placed in codearea or an instruction placed in dataarea. Since multiple labels (many labels preceding the same instruction can occour) it is nescessary to chain these declarations together until the instruction is reached.

When a label is met it is placed in a chain starting at the simple variable labelstart (the chain is kept in dectabarea).

When the type of the instruction (codearea or dataarea) is known the assembler works its way through this chain and puts a pointer to the declarationplace into each element of this chain.

Labelreferences can be met in codearea, dataarea or globarea. Insertref inserts the address, where the labeled was referenced, in a one-way-chain starting at reftabarea.labelnumber and chaining all references to this labelnumber,

The first 2 bit in the words in this pointerchain indicates in which area the address is:

bit 15

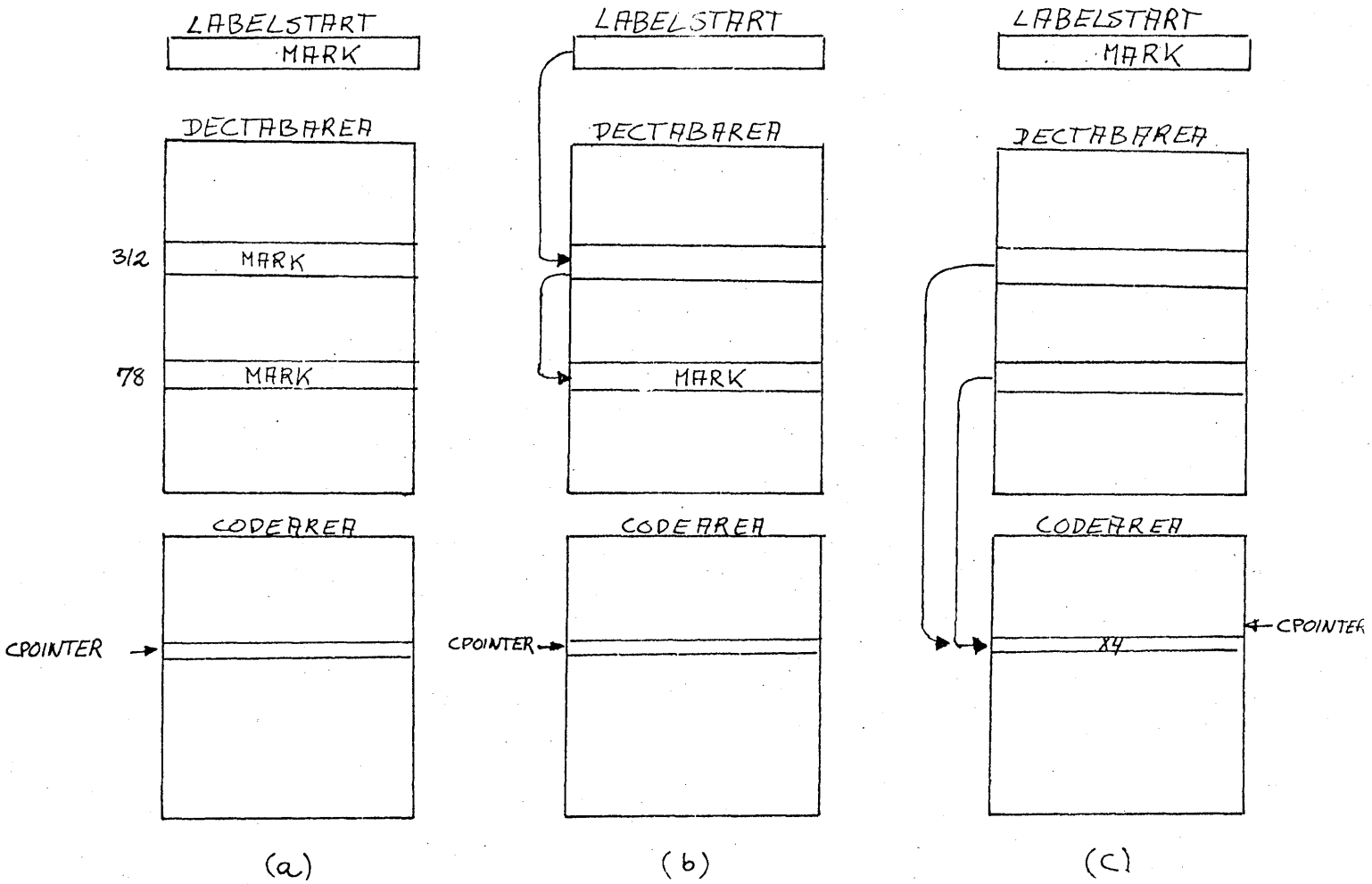
1 : codearea (bit 14-0 determines address)

0 : iff bit 14 = 1 : dataarea (bit 13-0 determines
0 : globarea address)

The pointerchain linking the labelreferences is kept in the cells in codearea, dataarea and globalarea where the intcode-loader shall put the final address (main store address).

As an endmark of such a chain is used mark (1111111111111101). Before start of a segment all cells in dectabarea and reftabarea are initialized to this mark.

See figures at page 2.14 and 2.15



This page and the next shows how labels are handled.
The corresponding instructions could be:

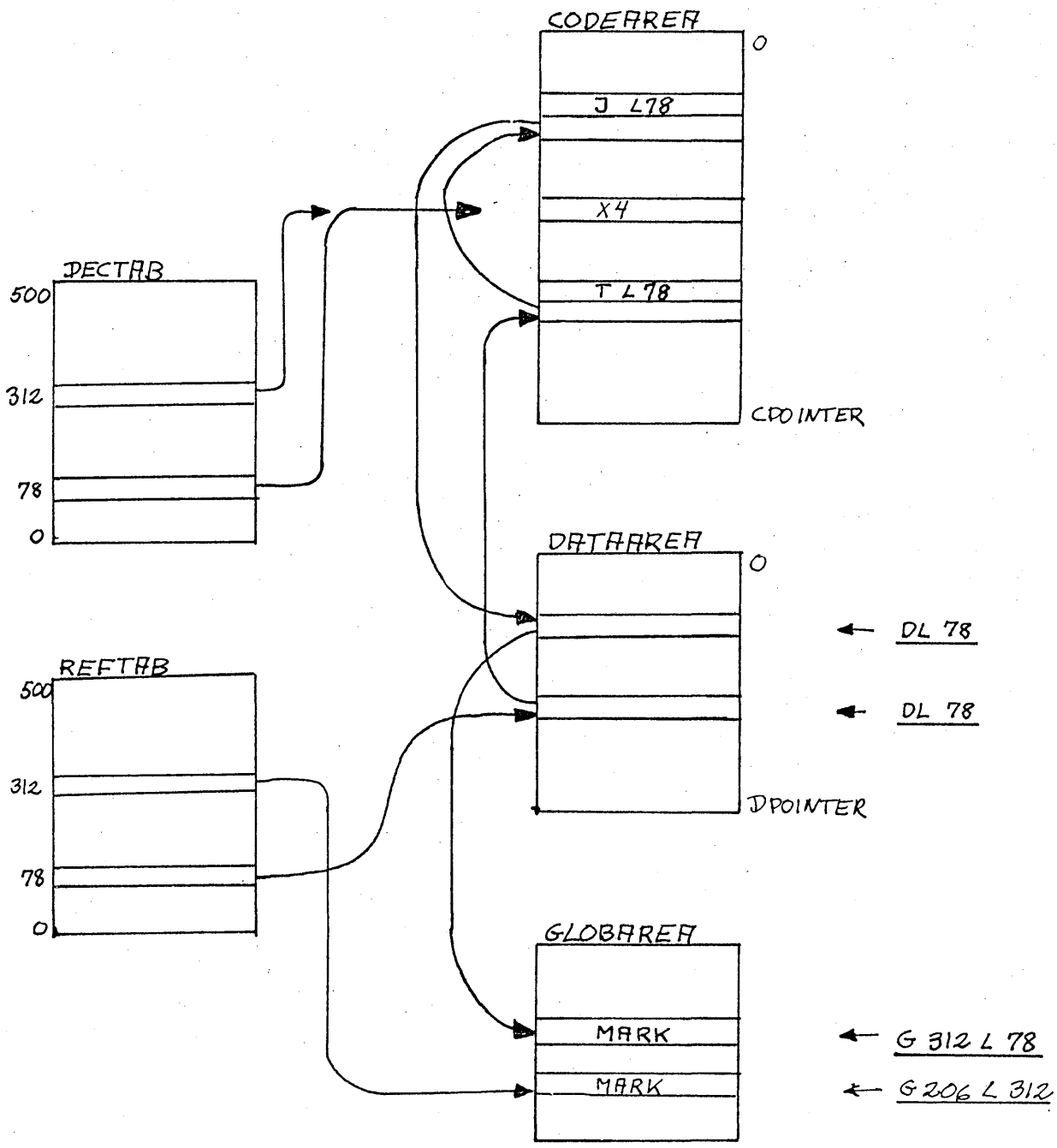
G206L312 G312L78 DL78 JL78 78 312 X4 TL78 JL78 .....

a) just before the declaration of label 78

b) both labeldeclaration has been read and chained

c) the instruction type is known and pointers from dectabarea to the address of declaration is made.

d) the final picture handed to the loader (including reftabarea)


CODEAREA
0
J L78
X4
T L78
CPOINTER
DECTAB
500
312
78
0
DATAAREA
0
DL 78
DL 78
DPOINTER
REFTAB
500
312
78
0
GLOBAREA
MARK
MARK
G 312 L 78
G 206 L 312

(d)

SWITCH- INSTRUCTION

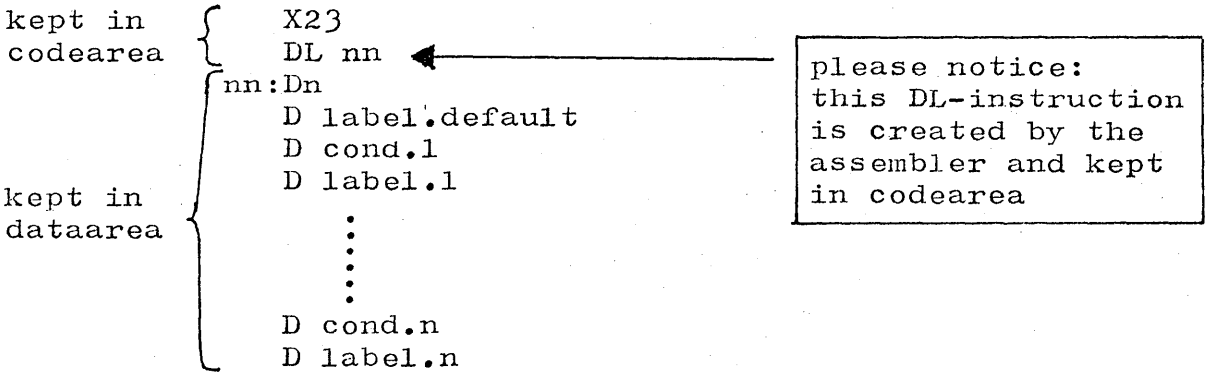
Instruction X23 is a switch instruction. According to the intcode discription it should be used as follows:

```
X23
Dn
DL label.default
D cond.1
DL label.1
D cond.2
DL label.2
    :
    :
    :
D cond.n
DL label.n
```

For the semantic explanation of this please see at page 1.6.

It is important to know that X23 is a normal-instruction and therefore kept in codearea. Its data (Dn ..... DL label.n) is D-instructions and hence kept in dataarea. It is nescessary to tell the X23-instruction where its data is.

Hence the above peace of code by the assembler is altered to:



where nn is a new label generated by the assembler.

By this reason no segment can have more than 1oo X23-instructions.

ERRORMESSAGES FROM ASSEMBLER

ADDRESS MISSING	normal-instruction has no address field
ADDRESS NEG OR TOO BIG	normal instruction with address outside {0 , 64K-1}
BAD LABELDECLARATION	labeldeclaration with labelnumber outside {1,500}
BAD LABELREFERENCE	labelreference with labelnumber outside {1,500}
C - BAD NUMBER	C-instruction with charvalue outside {0,255}
C - MISSING NUMBER	C-instruction has no charvalue
CODEAREA TOO BIG	the segment needs too much codearea
DATAAREA TOO BIG	the segment needs too much dataarea
DL- BAD LABELNO	DL-instruction with labelnumber outside {1,500}
DL- MISSING LABELNO	DL-instruction with "L" but no specified labelnumber
DOUBLEDECLARATION	labelnumber has been declared twice
D - MISSING NUMBER	D-instruction with no data
D - NUMBER TOO BIG	D-instruction with specified data outside {-32K , 32K-1}
ENDOFSTREAM REACHED	the end of source-text is reached at illegal point
G - BAD GLOBALNO	G-instruction with globalnumber outside {0,511}
G - BAD LABELNO	G-instruction with labelnumber outside {1,500}
G - L MISSING	G-instruction with "L" missing
G - MISSING GLOBALNO	G-instruction with no specified globalnumber
G - MISSING LABELNO	G-instruction with no specified labelnumber

ILLEGAL CHARACTER	illegal character used in sourcetext
LABEL PSEUDOINSTRUCTION	label is prefixing instruction which is neither a normal-instruction, D-instruction or C-instruction
LABELNUMBER MISSING	address field of normal-instruction has an "L" but no specified label-number
LABEL UNDECLARED: nnn	labelnumber nnn has been referenced but not declared (this errormessage is at the very last of the source-text-listing)
TOO MANY SWITCHES	the segment has more than 100 switch-instructions (X23 -instructions)

LOADER

INPUT

The loader takes input from the assembler.

The syntax is described on pages 2.9 through 2.11 and page 2.13.

STRUCTURE OF THE LOADER

The loader has this structure:

```
        declarations and initializations

        presentation

        initialization of cbase and dbase

    newtape:

        read until mark 1-0

        while nextoninput = mark 0-1 do

        §(while
            read header

            read code into mainstore

            read data into mainstore

            read globaltable and do the
            specified initializations

            read labeltable to initialize
            the unsolved references

            read mark 1-0

            sumcheck
        §)while

        if more tapes do goto newtape

        go()
```

(header = codesize, datasize, globalsize, tablesize and checksum)

THE USE OF GLOBALS G.504-511

To understand "go()" and "initialization of cbase and dbase" you must know that the system uses the globals 504 through 511 for communikation of the following information:

G.511	wlimit	-variable (program's limits and entrypoint)
G.510	p0	
G.509	C	
G.508	wlimit	-constant (system's limits and entrypoint)
G.507	p0	
G.506	C	
G.505	C	-loader's entrypoint
G.504	C	-assembler's entrypoint

INITIALIZATION OF CBASE AND DBASE takes G.508 as cbase(base of codearea) and G.507 as dbase (base of dataarea).

GO() initializes G.511, G.510 and G.509 with the loaded program's wlimit (cbase-1), p0 (dtop) and C (entrypoint).

Having done this it calles finish (see page 4.5).

SYSTEM

The system includes the assembler, the loader, the error-routine and a routine "system" to switch between loader and assembler.

ERROR (written in intcode - entrypoint=limit) is called from "error 1-5" in the emulator (see the last page in the emulatorlisting) to dump mainstore.0 to mainstore.15.
This pictures the registers (wa: 0-15) at the moment the error arose (except wa.3=errorno).

"SYSTEM" is written in intcode according to this algoritme:

```
        select(consoletable)
again:
        writes("*N assembler:a or loader:l ?")
        help:= input
        test help='A' then goto G.504
        else
        test help='L' then goto G.505
                      else goto again
```

(systems entrypoint is placed in G.506)

ERRORMESSAGES FROM LOADER

OWERFLOW	code- and dataarea too big
MISSING ENDMARK	the mark 1-0 expected after the segment is missing
BAD CHECKSUM	the checksum just calculated differs from the checksum given in the header

EMULATOR

ORDRECYCLUS

Emulating one intcode-instruction the emulator goes round once in the below mentioned cycle:

LOOP

1. Ask whether the surrounding system wants control (has issued an interrupt).
2. Read next instruction from mainstore to DS (doubleshifter).
3. Read address to AS (acumulatorshifter) and D. Iff L-bit it set the address is found in the next word in mainstore; else it is found in the last 9 bits of the present word.
4. Iff P-bit is set add P to D and keep the result in AS and D.
5. Iff G-bit is set add G to D and keep the result in AS and D.
6. Iff I-bit is set read the mainstore address D and put the content of this in AS and D.
7. Increment C (programcounter)
8. Decode instructioncode
9. According to this decoding jump to one of 8 possible labels in the emulator.
10. Perform the desired operation
11. goto 1.

There is one exception from this scheme. The X23 (switch-instruction) is so long that it must be partioned into smaller sections. Between each section the surrounding system has the possibility to interrupt.

"Flag" is used to indicate whether a switch-instruction is interrupted. Iff flag is set control is passed from 1. directly back to the part in the emulator handling X23-instructions.

INITIALIZATION OF WA AND WB

The first WB-group is initialized to:

0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, -1

The second WB-group is initialized to keep a table used by the emulator when switching between the 32 possible X-instructions.

0:	ERROR	address X1
1:	address X2	address X3
2:	address X4	address X5
	⋮	⋮
13:	address X26	address X27
14:	ERROR	ERROR
15:	ERROR	ERROR

The addresses (controlstoreaddresses) are packed two and two.

The first WA-group keeps the intcode-machines six registers and other vital information such as flags, limits etc.

0:	BASE
1:	WLIMIT
2:	PO
3:	ERROR
4:	FLAG
5:	B
6:	A
7:	C
8:	D
9:	P
10:	G
11:	
12:	INPUT
13:	OUTPUT
14:	CS
15:	LIMIT

BASE is the basisaddress in mainstore. All other addresses is relative to BASE.

WLIMIT is the last word in mainstore where writting is permitted.

PO is the first word in the runtimestack (local variables).

FLAG indicates whether a X23-instruction was interrupted (see page 4.1).

A, B, C, D, P and G are the intcode-machines six registers.

INPUT and OUTPUT keeps the selected input- and outputdevice.

CS is baseaddress in controlstore for the emulatorcode.

LIMIT is the last word in mainstore where access (reading) is permitted.

ERROR is a mailbox indicating where the errorroutine was called

BASE, PO, WLIMIT, LIMIT and C are initialized by the intcode-loader.

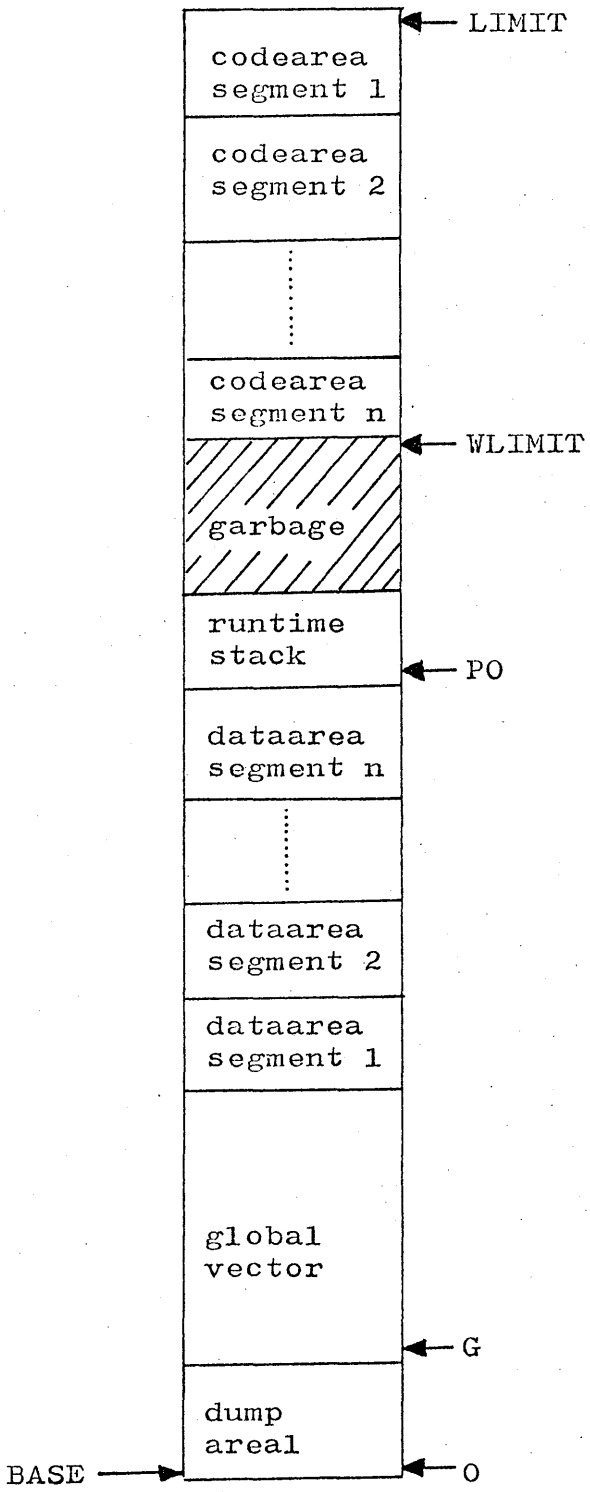
INPUT and OUTPUT has standard initializations (see microcode listing).

CS is initialized by the system (at present it's always 0).

Anythings else is initialized to 0.

MAINSTORE

When n segments are loaded the picture is:



READ AND WRITE

The emulator uses two subroutines: read and write.

These routines simply initialize the reading (writing). Then it returns control to the calling address without waiting for the read (write) to finish.

Read tests: base $\leq$ read-address $\leq$ limit

Write tests: base $\leq$ write-address $\leq$ wlimit

If these conditions are not met the errorroutine is called.

FINISH (X22 in the emulator)

X22 looks at G.511 to see if a program has been loaded (see page 3.2).

If so then it initializes wlimit,p0 and C from G,511, G.510 and G.509 and starts programexecution.

Else it initializes wlimit, p0 and C from G.508, G.507 and G.506 and so leaves control to "system".