INTCODE

documentation:

henrik andersen
kurt jensen
børge s. kirk
CONTENTS

1.1 INTCODE
1.1 store and registers
1.2 instruction formats
1.3 instruction codes
1.5 execute operations

2.1 ASSEMBLER
2.1 syntax - informal description
2.3 syntax - formal description
2.5 structure of assembler
2.9 output from assembler
2.12 working tables
2.13 label declarations and label references
2.16 switch - instruction
2.17 error messages from assembler

3.1 LOADER
3.1 input
3.1 structure of the loader
3.2 the use of globals G, 504-511
3.3 system
3.4 error messages from loader

4.1 EMULATOR
4.1 ordrecyclus
4.2 initialization of WA and WB
4.4 main store
4.5 read and write
4.5 finish (X22)
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 instruction formats. 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 global vector. Is used to reference elements in global vector.
Each instruction consists of **six fields** as follows:

**Instruction code:** 3 bits giving 8 possibilities

**Address field:** 9 or 16 bit (depending on 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 instruction format is used.

The **effective address** is calculated in 4 steps:

1. \( D := \) address field (depending on L-bit)
2. Possible adding of P-register
3. Possible adding of G-register
4. Possible indirectness

```
 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
SHORT: code P G I 0 address
LONG: code P G I 1 garbage
       address
```
<table>
<thead>
<tr>
<th>Code</th>
<th>Mnemonic</th>
<th>Description</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>Load</td>
<td>$B := A; A := D$</td>
<td>This is not a &quot;normal&quot; load-instruction. It loads the effective address and not the content of this address.</td>
</tr>
<tr>
<td>001</td>
<td>Store</td>
<td>$\text{loc}(D) := A$</td>
<td></td>
</tr>
<tr>
<td>010</td>
<td>Add</td>
<td>$A := A + D$</td>
<td></td>
</tr>
<tr>
<td>011</td>
<td>Jump</td>
<td>$C := D$</td>
<td>Unconditional jump to absolute address $D$.</td>
</tr>
<tr>
<td>100</td>
<td>Jump if True</td>
<td>if $A = 0$ then $C := D$</td>
<td>Conditional jump to absolute address $D$.</td>
</tr>
<tr>
<td>101</td>
<td>Jump if False</td>
<td>if $A \neq 0$ then $C := D$</td>
<td></td>
</tr>
<tr>
<td>110</td>
<td>K</td>
<td>$\text{loc}(P+D) := P$ $\text{loc}(P+D+1) := C$ $P := P+D$ $C := A$</td>
<td>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.</td>
</tr>
<tr>
<td>111</td>
<td>Execute</td>
<td></td>
<td>Allows auxiliary operations to be executed. The operation is specified by $D$ modulo 32 (last five bits).</td>
</tr>
</tbody>
</table>
Figure showing the effect of a K-instruction

BEFORE:

data area n+2

Return Address

data area n+1

Return Address

data area n

AFTER:

data area n+3

A

P

data area n+2

Return Address

data area n+1

Return Address

data area n

Reg. P

Reg. P
### EXECUTE OPERATIONS

<table>
<thead>
<tr>
<th>No.</th>
<th>Description</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>X1</td>
<td>A := loc(A)</td>
<td>Return from current routine or function.</td>
</tr>
<tr>
<td>X2</td>
<td>A := -A</td>
<td>Result of function is left in A.</td>
</tr>
<tr>
<td>X3</td>
<td>A := not A</td>
<td>Integer division</td>
</tr>
<tr>
<td>X4</td>
<td>C := loc(P+1)</td>
<td>Rest after integer division.</td>
</tr>
<tr>
<td></td>
<td>P := loc(P)</td>
<td></td>
</tr>
<tr>
<td>X5</td>
<td>A := B * A</td>
<td></td>
</tr>
<tr>
<td>X6</td>
<td>A := B / A</td>
<td></td>
</tr>
<tr>
<td>X7</td>
<td>A := B rem A</td>
<td></td>
</tr>
<tr>
<td>X8</td>
<td>A := B + A</td>
<td></td>
</tr>
<tr>
<td>X9</td>
<td>A := B - A</td>
<td></td>
</tr>
<tr>
<td>X10</td>
<td>A := B = A</td>
<td></td>
</tr>
<tr>
<td>X11</td>
<td>A := B ≠ A</td>
<td></td>
</tr>
<tr>
<td>X12</td>
<td>A := B &lt; A</td>
<td>These operations yield a boolean result (true or false)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X13</td>
<td>A := B ≥ A</td>
<td></td>
</tr>
<tr>
<td>X14</td>
<td>A := B &gt; A</td>
<td></td>
</tr>
<tr>
<td>X15</td>
<td>A := B ≤ A</td>
<td></td>
</tr>
<tr>
<td>X16</td>
<td>A := B lshift A</td>
<td>B is shifted A times logical left</td>
</tr>
<tr>
<td>X17</td>
<td>A := B rshift A</td>
<td>B is shifted A times logical right</td>
</tr>
<tr>
<td></td>
<td>&quot;B lshift A&quot; is equivalent to &quot;B rshift A&quot;</td>
<td></td>
</tr>
<tr>
<td>X18</td>
<td>A := B ∧ A</td>
<td></td>
</tr>
<tr>
<td>X19</td>
<td>A := B ∨ A</td>
<td></td>
</tr>
<tr>
<td>X20</td>
<td>A := B ⊕ A</td>
<td></td>
</tr>
<tr>
<td>X21</td>
<td>A := B ≡ A</td>
<td></td>
</tr>
<tr>
<td>X22</td>
<td>finish</td>
<td></td>
</tr>
</tbody>
</table>
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
collection.2
label.2
...
condition.2
label.n

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

The above piece 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

selects input device according to A
input data to A from selected input device
selects output device according to A
output data from A to selected output device
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 or more segments each consisting of a sequence of intcode-instructions terminated by a special Z-instruction.

Instructions can be of 6 different kinds:

- normal-instruction: 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 consists 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 absolute address (integer) or a label.

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

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 character values in data area. Each character value is prefixed by a C.

The character values 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 label declaration the right half will be padded with zeros.

G-instruction: is used to initialize element of global vector with the value of a label number.

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

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

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

Z-instruction: terminates a segment and cannot be used anywhere else.
The syntax is now described in ursIF.

Underlined symbols are terminals. 

{n,m} means an integer in the closed interval \([n,m]\)

Everywhere in the source text / can be inserted - then this charac-
ter 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.

\[
\begin{align*}
\langle \text{program} \rangle & ::= \langle \text{segment} \rangle^+ \\
\langle \text{segment} \rangle & ::= \langle \text{instruction} \rangle^* \langle \text{Z-instruc} \rangle \\
\langle \text{instruction} \rangle & ::= \langle \text{normal-instruc} \rangle | \\
& \quad \langle \text{D-instruction} \rangle | \\
& \quad \langle \text{C-instruction} \rangle | \\
& \quad \langle \text{G-instruction} \rangle | \\
& \quad \langle \text{L-instruction} \rangle \\
\langle \text{normal-instruc} \rangle & ::= \langle \text{labelpart} \rangle \langle \text{instruccode} \rangle \langle \text{IPG-bit} \rangle \langle \text{address} \rangle \\
\langle \text{D-instruction} \rangle & ::= \langle \text{labelpart} \rangle \langle \text{data} \rangle \\
\langle \text{C-instruction} \rangle & ::= \langle \text{labelpart} \rangle \langle \text{charvalue} \rangle \\
\langle \text{G-instruction} \rangle & ::= \langle \text{skip} \rangle \langle \text{Global} \rangle \langle \text{L} \rangle \langle \text{label} \rangle \\
\langle \text{L-instruction} \rangle & ::= \langle \text{skip} \rangle (Y | N) \\
\langle \text{Z-instruction} \rangle & ::= \langle \text{skip} \rangle Z \langle \text{skip} \rangle \\
\langle \text{labelpart} \rangle & ::= \langle \text{skip} \rangle (\langle \text{label} \rangle \langle \text{delim} \rangle)^* \\
\langle \text{skip} \rangle & ::= (\# \# \# | \$)^* \\
\langle \text{delim} \rangle & ::= (\# \# \# | \$)^* \\
\langle \text{label} \rangle & ::= \{1,500\} \\
\langle \text{IPG-bit} \rangle & ::= (I | P | G)^* \\
\langle \text{address} \rangle & ::= \{0, 64K-1\} | \langle \text{label} \rangle \\
\langle \text{instruccode} \rangle & ::= (L | S | A | I | T | F | K | X)\
\end{align*}
\]
\[ \text{<data>} ::= \{-32K, 32K-1\} \quad \text{and} \quad \text{<charvalue>} ::= \{0,255\} \]
\[ \text{<global>} ::= \{0,511\} \]
STRUCTURE OF ASSEMBLER

The program is divided into 3 parts

asshead keeps all global and manifest declarations
assproc keeps all procedures
assmain keeps the main program

assproc and assmain use 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

\[
\text{switchon ch into}
\]

\[
\text{case L:}\quad \text{goto instruction}
\]

\[
\text{case X:}
\]

\[
\text{case $,\star n,\star s:}
\]

\[
\text{case D:}
\]

\[
\text{case C:}
\]

\[
\text{case G:}
\]

\[
\text{case Y:}
\]

\[
\text{case N:}
\]

\[
\text{case Z:}\quad \text{goto newsegment}
\]

\[
\text{default:}\quad \text{goto endcase}
\]

instruction:

\[
\text{endcase:}
\]

\[
\text{$)mainloop repeat}
\]
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
insruct: read()

switchon ch into

$(switch2
case I:
  goto endcase2
case P:    
  goto instruc1
endcase2:
case G:    
  goto instruc1

default: test ch = L
  then
      address specified as a labelvalue
  or
      $( address specified as an absolute address
      check if address is a X23 instruction (see page 2.16)

})switch2
```

case S,m,*: handles insignificant characters. These are skipped.
case D: handles D-instructions. The structure is:

```plaintext
case D: read()

switch on 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
```

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

```plaintext
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:

```plaintext
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)
```
\textbf{case Z:} handles Z-instructions. This signals the end of a segment. The structure is:

\begin{verbatim}
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 $( mark 1-0
        mark 1-0
        closeall()
        finish
    )$
\end{verbatim}

\textbf{default:} handles labeldeclarations (or illegal sourcetext characters). The structure is:

\begin{verbatim}
default: while ch = tal
    $(
        collect labelnumber (declaration)
        update dectabarea (see page 2.13)
    )$

\textbf{switch on} ch \textbf{into}

\begin{verbatim}
$(sw
    case D:  } declaration-address in dataarea
    case C:  
    case L:  }
    case X:  
\end{verbatim}

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

\begin{verbatim}
goto endcase (see bottom of page 2.5)
\end{verbatim}

\begin{verbatim}
case G, Z, Y, N: illegal labeldeclaration
goto endcase
\end{verbatim}

\begin{verbatim}
default: illegal sourcetext character
    goto endcase
\end{verbatim}
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.

```
segmentblock for first correct segment
    ...
segment block for last correct segment
    mark 1-0
    mark 1-0
```

where `segmentblock` is

```
mark 1-0
mark 0-1
size of codearea
size of dataarea
number of globals to initialize
number of referenced labels
sumcheck
picture of codearea
picture of dataarea
globaltable
labeltable
```

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 tables (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
...
...
```

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 initialized; 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.
Labeltable 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.e. labeldec and i. labelref) from which all necessary 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 (111111111111101).

The number of the label is insignificant and is not given in the output of the assembler.
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 label-number. 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 reftarea handles assembler generated labels (see p 2.10). Lpointer points to the first free labelname in decarea and reftarea.

When a correct segment is finished 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 "globaltable".

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 labeltable.

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

The number of referenced labels in reftabarea and reftarea is dumped as "number of referenced labels".
When a label declaration is met it is necessary to know whether it is preceding an instruction placed in code area or an instruction placed in data area. Since multiple labels (many labels preceding the same instruction can occur) it is necessary 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 dectab area).

When the type of the instruction (code area or data area) is known the assembler works its way through this chain and puts a pointer to the declaration place into each element of this chain.

Label references can be met in code area, data area or glob area. Insertref inserts the address, where the labeled was referenced, in a one-way chain starting at reftab area, label number and chaining all references to this label number.

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

bit 15

1 : code area (bit 14-0 determines address)

0 : iff bit 14 = 1 : data area (bit 13-0 determines 0 : glob area address)

The pointer chain linking the label references is kept in the cells in code area, data area and global area where the int code loader shall put the final address (main store address).

As an endmark of such a chain is used mark (111111111111101). Before start of a segment all cells in dectab area and reftab area 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 label declaration 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)
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 necessary to tell the X23-instruction where its data is.

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

```
kept in codearea
  { X23
    DL nn
    nn:Dn
    D label.default
    D cond.1
    D label.1
    ...
    ...
    D cond.n
    D label.n
  }

kept in dataarea
```

please notice: this DL-instruction is created by the assembler and kept in codearea

where nn is a new label generated by the assembler.

By this reason no segment can have more than 100 X23-instructions.
ERROR MESSAGES 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
<table>
<thead>
<tr>
<th>Error Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ILLEGAL CHARACTER</td>
<td>Illegal character used in source text</td>
</tr>
<tr>
<td>LABEL PSEUDOINSTRUCTION</td>
<td>Label is prefixing instruction which is neither a normal-instruction, D-instruction or C-instruction</td>
</tr>
<tr>
<td>LABELNUMBER MISSING</td>
<td>Address field of normal-instruction has an &quot;L&quot; but no specified label-number</td>
</tr>
<tr>
<td>LABEL UNDECLARED: nnn</td>
<td>Label number nnn has been referenced but not declared (this error message is at the very last of the source-text-listing)</td>
</tr>
<tr>
<td>TOO MANY SWITCHES</td>
<td>The segment has more than 100 switch-instructions (X23-instructions)</td>
</tr>
</tbody>
</table>
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 communication of the following information:

- **G.511**: wlimit -variable (program's limits and entrypoint)
- **G.510**: p0 -variable (program's limits and entrypoint)
- **G.509**: C -variable (program's limits and entrypoint)
- **G.508**: wlimit -constant (system's limits and entrypoint)
- **G.507**: p0 -constant (system's limits and entrypoint)
- **G.506**: C -loader's entrypoint
- **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 calls 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 - entypoint=limit) is called from "error 1-5" in the emulator (see the last page in the emulator listing) to dump mainstore.0 to mainstore.15. This pictures the registers (wa: 0-15) at the moment the error arose (except wa.3=errno).

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

```
select(consoletable)
again:
    writes("IN 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

OVERFLOW  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:

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

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 partitioned 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 l. directly back to the part in the emulator handling X23-instructions.
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.

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>ERROR</td>
<td>address X1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>address X2</td>
<td>address X3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>address X4</td>
<td>address X5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>address X26</td>
<td>address X27</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>ERROR</td>
<td>ERROR</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>ERROR</td>
<td>ERROR</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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.

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0:</td>
<td>BASE</td>
</tr>
<tr>
<td>1:</td>
<td>W LIMIT</td>
</tr>
<tr>
<td>2:</td>
<td>PO</td>
</tr>
<tr>
<td>3:</td>
<td>ERROR</td>
</tr>
<tr>
<td>4:</td>
<td>FLAG</td>
</tr>
<tr>
<td>5:</td>
<td>B</td>
</tr>
<tr>
<td>6:</td>
<td>A</td>
</tr>
<tr>
<td>7:</td>
<td>C</td>
</tr>
<tr>
<td>8:</td>
<td>D</td>
</tr>
<tr>
<td>9:</td>
<td>P</td>
</tr>
<tr>
<td>10:</td>
<td>G</td>
</tr>
<tr>
<td>11:</td>
<td></td>
</tr>
<tr>
<td>12:</td>
<td>INPUT</td>
</tr>
<tr>
<td>13:</td>
<td>OUTPUT</td>
</tr>
<tr>
<td>14:</td>
<td>CS</td>
</tr>
<tr>
<td>15:</td>
<td>LIMIT</td>
</tr>
</tbody>
</table>

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

WLIMIT is the last word in mainstore where writing 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.
When \( n \) segments are loaded the picture is:

```
BASE 0

segment 1
segment 2
segment n

LIMIT
WLIMIT

garbage
runtime stack
PO

dataarea segment n
dataarea segment 2
dataarea segment 1

global vector

dump area

G
O
```
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 error routine 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 program execution.

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