Paul McJones 1964

COOL-AID

Contents

P.57 [illegible]

P16 [illegible]

Change [illegible] throughout

0. Format

The assembler uses an infix, algebraic language-style format. Blanks may optionally appear between words and delimiters, and must appear between two directly adjacent words. Statements are terminated with a semicolon or carriage return character. A statement may be broken across two or more lines - an occurrence of ' ...Q' followed by carriage return is equivalent to a blank. Thus words and composite operators (such as >=) may not be split across lines.

no blank space }

Statement [illegible] any string of characters CR = statement CR

statement [illegible] any string [illegible] CR continuation of statement

≡ statement

D and a scale factor, then the value of the number is the value it would have if the scale-factor were absent, multiplied by [radix ↑ s], where radix is 2 or 10 , for respectively a B or D integer, and s is the base 10 value of the scale factor. A number containing a decimal point $\overset{r}{\text{of}}$ an E represents a value in machine floating-point format. The possibly signed scale-factor indicates the power of ten by which the rational number before the E is multiplied.

String constants may be of any length, although when a string appears in an expression, characters past the eighth (if any) will be truncated. The value of a string is a series of 7 bit bytes (each representing a character coded in internal ASCII). When used in expressions, the bytes will normally be right-adjusted in a word, with the unused high order positions containing zeros. However, a string constant followed by an L forces the bytes to be positioned against the left, high-order end of the word.

1.1.2. Symbols symbol ::= { $ • { letter [{letter | digit} ••• } ••• }

1.1.3. Counters

a symbol is

1.1.3.1. Syntax

origin-counter ::= *Ø

location-counter ::= *[L]

position-counter ::= *P

counter ::= *origin-counter* | *location-counter* | *position-counter*

1.1.3.2. Semantics

The assembler maintains three counters, which determine the current place in the object program being created. The origin counter specifies at what address instructions and data are to be loaded into the computer's memory. Whenever a complete word of the object program is formed, the origin counter is incremented by one. Several statements described in section 2.4 also affect its value, which must lie in the interval $0 \leq$ *Ø $\leq$ $\overset{3}{\text{B}}$77777B .

Representing the number of bits remaining in the current word being assembled in which data or instructions may be placed, the position counter always has a value in the range $60 \geq$ *P ≥ 1 . Thus before each of the instructions of a word containing four 15-bit instructions, *P would have the values 60 , 45 , 30 , and 15 .

The location counter normally has the same value as the origin counter, since it is also incremented by 1 whenever a word of the object program is completed. However, use of the statements in section 2.4 can make these counters differ. Since labels on assembly statements are defined by the assembler with the current value of the location counter (cf. section 2), the ability to control the location counter facilitates writing programs which will be loaded at one point (governed by the origin counter) but are ultimately to execute at another point (dictated by the location counter). Values of the location counter must fall in the interval $0 \leq$ *L $\leq$ 377777B .

1.2. Operators of assembly expressions

1.2.1. Syntax

primary ::= primitive | {+|-} primary | [expr]

[illegible]

factor ::= primary | primary ↑ factor

term ::= factor | term {*|/|\\} factor

sum ::= term | sum {+ | -} term

relation ::= sum [{< | <= | = | # | <= | <} sum]

negation ::= relation | ? negation

conjunction ~~disjunction~~ ::= negation | conjunction ~~disjunction~~ & negation

disjunction ~~conjunction~~ ::= conjunction ~~disjunction~~ | disjunction ~~conjunction~~ {! | ?} disjunction

expr ~~literal~~ ::= disjunction ~~conjunction~~ | = expr ~~literal~~

~~expr ::= literal | literal \ expr~~

1.2.2. Examples

Primaries:

1B59

APPLE

*Ø

-0

[['X' & 17B] ↑ [2 * 14]]

Factors:

'01234567'L

RADIX ↑ SCALE

Terms:

INTEGER * RADIX ↑ SCALE

[EXTERNAL + 140B] \\ 200B

Sum:

TABLE + 2 * INDEX + 1

Relations:

--SYMBOL

COUNT < 15

```
Negation:
  ??UNNORMALIZEDBOOLEAN
Di^sjunction:
  0 <= * & * < 2 ↑ 17 - 1
Conjunction:
  1 ! 10B ! 1B6 ! 1B9 ? 3.14159
~~Literals~~: Expls:
  A * B
  = 1.0
2 ==P1ASTERISK-P1BASE
~~Expressio~~ns:
2 BYTESIZE \ [H > 0] * X + [H <= 0] * Y
3 30\[= N * ITEMSIZE + HEADERLENGTH
```

1.2.2. Semantics

Expressions are evaluated by associating with each primary its value, then combining these values in the order specified by the syntax. In particular, a subexpression enclosed by square brackets will be fully evaluated before the evaluation of the expression within which it occurs is completed. The syntax of section 1.2.1. indicates that the following rules of precedence control the order of evaluation:

```
highest: unary + and -
         ↑
         *, /, and \\
         + and -
         <, <=, =, #, >=, and >
         unary ?
         &
         ! and ?
         unary =
lowest:  \\
```

Operators with the same precedence are usually applied from left to right, except for the unary operators and the binary operators ↑ and \ , which are applied from right to left. All operators but \ and : produce a result whose *length attribute* is 60 , while the result of the \ operator has a length attribute which is the value of the left operand, and the length of the result of : is the sum of the length attributes of the operands. The length attribute is of interest in the expression lists of DATA statements and symbol assignment statements (cf. sections 2.2.3. and 2.3.3.). Operators may be classified as arithmetic, relational, boolean, or miscellaneous.

1.2.~~2~~3.1. Arithmetic operators

These operators, including +, -, *, /, \\\\ , and ↑ , and representing addition, subtraction, multiplication, division, remaindering, and exponentiation, treat their operands as signed integers. If a division produces a remainder, then the *magnitude* of the result will be truncated to the nearest smaller integer. The value of A\\\\B is defined in terms of division; it is A-A/B*B . The left operand of an exponentiation is the base, and the right operand is the exponent. Thus 2↑3↑2 has the value $2^{(3^2)}$, or 512, while the value of [2↑3]↑2 is $(2^3)^2$ or 64 . The value of A↑B is given by the following rules:

```
      If   B > 0  then  A*A*···*A  (B  times)
else if   B = 0  then if  A ≠ 0  then  1
                 else error
else if   A ≠ 0  then  0
else  error
```

1.2.~~2~~3.2. Relational operators

The relational operators <, <=, =, #, >=, and >, where the '#' represents "not equal" and the composite operators '<=' and '>=' represent

$$\text{do-stmt} ::= \quad \text{DØ} \left[\begin{array}{l} \text{A-reg} \left[\leftarrow \text{set-oprnd}\right] \left[\text{BY} \left\{\begin{matrix}\text{expr} \\ [-]\text{B-reg}\end{matrix}\right\}\right] \\ \text{B-reg} \left[\leftarrow \text{set-oprnd}\right] \left[\text{BY} \left\{\begin{matrix}\text{expr} \\ \text{A-res} \\ [-\text{B-res}] \\ \text{X-res}\end{matrix}\right\}\right] \\ \text{X-res} \left[\leftarrow \text{set-oprnd}\right] \left[\text{BY} \left\{\begin{matrix}\text{expr} \\ \text{B-res}\end{matrix}\right\}\right] \end{array}\right] \left[\text{WHILE cond}\right]$$

$$\left[\text{stmt} \bullet\bullet\bullet\right] \quad \text{ENDØ}$$

$$\begin{array}{lll} \text{if-stmt} & ::= & \text{if-clause stmt} \mid \text{if-clause bal-stmt ELSE stmt} \\ \text{bal-stmt} & ::= & \text{uncond-stmt} \mid \text{if-clause bal-stmt ELSE bal-stmt} \\ \text{if-clause} & ::= & \text{IF cond THEN} \end{array}$$

$$\text{cond} ::= \left\{\begin{array}{l} \text{X-reg} \{= \mid \# \mid >= \mid < \} \, 0 \\ 0 \{= \mid \# \mid <= \mid > \} \, \text{X-reg} \\ \{\text{IR} \mid \text{ØR} \mid \text{DF} \mid \text{ID}\} \, \text{X-reg} \\ \text{B-reg} \{= \mid \# \mid >= \mid < \mid <= \mid > \} \{\text{B-reg} \mid 0\} \\ 0 \{= \mid \# \mid >= \mid < \mid <= \mid > \} \, \text{B-reg} \end{array}\right\}$$

IR ≡ FINITE
ØR ≡ INFINITE
DF ≡ DEFINITE
ID ≡ INDEF

$$\text{set-oprnd} ::= \left\{\begin{array}{l} \left\{\begin{matrix}\text{A-res} \\ \text{B-reg} \\ \text{X-reg}\end{matrix}\right\} + \text{K} \\ \text{K} \left[+ \left\{\begin{matrix}\text{A-reg} \\ \text{B-res} \\ \text{X-reg}\end{matrix}\right\}\right] \\ \text{X-res} \, [+\text{B-res}] \mid \text{B-res} + \text{X-reg} \\ \{\text{A-res} \mid \text{B-res}\} \, [\pm \text{B-reg}] \mid \text{B-res} + \text{A-res} \\ -\text{B-reg} \, [+ \{\text{Ares} \mid \text{B-res}\}] \end{array}\right\}$$

$$\text{K} \equiv \text{expr}$$

"less than or equal" and "greater than or equal" , take arithmetic operands and return the 60 bit strings 000···01 and 000···00 , according to whether the indicated relation is *true* or *false*, respectively.

1.2.$\overset{3}{\cancel{2}}$.3. Boolean Operators

In this group is the unary operator '?' and the binary operators '&', '!', and '?' . Unary '?' is the negation operator, mapping +0 to 1 , and all other operands to +0 . By the definition of section 1.2.2.2., it maps *false* to *true* and "not-*false*" to *false*, since its domain is larger than its range.

The three binary operators '&', '!', and '?' , denoting AND , inclusive ØR , and exclusive ØR, treat their operands as bit strings of length 60. Each bit of the result is computed from the corresponding bits of the operands with the particular boolean function.

The 60-bit ones-complement of a number is obtained by the unary '-' operator, while the unary '?' negation operator is used with truth values. ??X "normalizes" X , that is its value will be only 0 or 1 , so that boolean expressions, particularly those using '&' , will produce the desired value.

1.2.$\overset{3}{\cancel{2}}$.4. Miscellaneous Operators

The value returned by the literal operator, unary = , is the address of a word containing the value of the expression which is the operand of the literal operator. A pool of literals is maintained by the assembler in a use-block named LITERALS (cf. section 2.4.3.). Each use of the literal operator causes an entry to be added to the pool if none existed with a value equal to the operand expression. The value of a literal expression is the address in the literal pool of the word containing the expression, and is always forward defined since the start of the LITERALS block is unknown until the end of assembly.

In order to facilitate packing information in DATA and symbol assignment statements (cf. sections 2.2.3. and 2.3.3.), the length operator \ is provided. It produces a result whose value is the right operand and whose length attribute is the left operand. Thus the expression A\B implies that the value of B will fit in a field A bits long. A and B must satisfy the relation A > FLOOR(LOG2(ABS(B))).

2. Assembly Statements

assembly-stmt ::= [*label-part*] {*machine-instr* | *data-stmt* | *symbol-asgn-stmt* | *misc-stmt*}

label-part ::= {*symbol*:} ··· | {+ | -}:

The statements described in this section are directly related to the fundamental assembly process. They generate machine instructions and data words in the binary output file, define symbols with values which may be used in expressions, and change the internal state of the assembler, thus governing the effect of subsequent statements.

In addition to the three counters described in section 1.1.3.2., part of the status maintained by the assembler is the *force upper flag*, which takes part in determining whether the next information to be assembled will start left adjusted in a new word. This flag is a "trit" , having one of the three values *false, don't care,* or *true* . Each machine instruction and data generating statement leaves the flag in the *true* or *don't care* state. After processing the label-part, which may affect the force upper flag, the assembler forces upper if the force upper flag is *true* . Forcing upper is equivalent to executing the following assembly statement (cf. section 2.2.3.):

```
DATA  *P\\15 \ 0;[*P=45]*15 \ 46000B; ...
          [*P=30]*15 \46000B;[*P=15]*15 \ 46000B
```

The effect is to fill out the current word with no-op instructions (unless *P already equals 60) after filling out to the nearest quarter-word boundary with zero-bits, if *P modulo 15 is not already 0 . Certain machine instructions will automatically be forced upper even if the flag is set to *don't care* (cf. section 2.1.2.). If the force upper flag is *false* , the assembler will force upper only if the item to be assembled will not fit in the current word.

Any assembly statement may contain a label-part, which will result in the following actions taking place before the statement itself is executed. If the label-part consists of a $\overset{\text{plus}}{+}$ or a minus followed by a colon, then the force $\overset{\text{offset}}{\wedge}$ flag will be set to *true* or *false* respectively, regardless of how the flag was previously set. If the label-part consists of a sequence of symbols each followed by a colon, then first/the assembler forces upper. Then each of the symbols, called labels in this context, is defined with the current value of the location counter. For each symbol $\overset{S}{L_i}$ of the label-part, the effect is as if the following statement were executed (cf. section 2.3.3.):

$$\overset{S}{L_i} = *L$$

2.1. Machine Instructions

2.1.1. Syntax

machine-instruction ::=

```
        PS   [expr]    |

        RJ   expr     |

       (RE)
       (WE) (B-reg [{+ | -} expr])
       (XJ) (expr [+ B-reg]     )    |
       (JP)

             (X-reg {= | # | >= | <} 0                                 )
             (0 {= | # | <= | >} X-reg                                 )
        IF   ({INRANGE | OUTRANGE | DEFINITE | INDEFINITE} X-reg       ) GOTO expr |
             (B-reg {= | # | >= | < | <= | >} {B-reg | 0}              )
             (0 {= | # | <= | > | >= | <} B-reg                        )
        ^

             ( [-] X-reg [{& | ! | ?} X-reg]                                      )
             (  0                                                                 )
             ( (LCYCLE)                                                           )
             ( (RCYCLE)                                                           )
             ( (RSHIFT) expr                                                      )
             ( (MASK  )                                                           )
             ( ([illegible])                                                      )
             (                                                                    )
  X-reg =    (           (LCYCLE)                                                 )  |
             (           (RSHIFT)                                                 )
             (           (NORM  )                                                 )
             ( [B-reg]   (RNORM ) X-reg                                           )
             (           (UNPACK)                                                 )
             (           (PACK  )                                                 )
             (                                                                    )
             ( X-reg {{+. | -. | +.. | -.. | .+. | .-. | + | - |                  )
             (          *. | .*. | *.. | /. | ./.} X-reg                          )
             (                                                                    )
             ( COUNT  X-reg                                                       )

             X-reg  (LCY   )
                    (RSH   )
                    (NORM  )  B-reg
                    (RNORM )
                    (UNPACK)
                    (PACK  )
```

"X-reg" should be under "IF"

move to right $\Rightarrow$

$$\left\{\begin{matrix}\textit{A-reg}\\ \textit{B-reg}\\ \textit{X-reg}\end{matrix}\right\} \leftarrow \left\{\begin{array}{l}
[+] \left\{\begin{matrix}\textit{A-reg}\\ \textit{B-reg}\\ \textit{X-reg}\end{matrix}\right\} \{+ \mid -\}\ \textit{expr}\\[1em]
\qquad \textit{expr}\ [+ \left\{\begin{matrix}\textit{A-reg}\\ \textit{B-reg}\\ \textit{X-reg}\end{matrix}\right\}\ ]\\[1em]
[+]\ \textit{X-reg}\ [+\ \textit{B-reg}]\\
[+]\ \textit{B-reg}\ +\ \{\textit{X-reg} \mid \textit{A-reg}\}\\
[+]\ \{\textit{A-reg} \mid \textit{B-reg}\}\ [\{+ \mid -\}\ \textit{B-reg}]\\
\ -\ \ \textit{B-reg}\ [+\{\textit{A-reg} \mid \textit{B-reg}\}]
\end{array}\right\}$$

A-reg	::=	A	octal-digit
B-reg	::=	B	octal-digit
X-reg	::=	X	octal-digit

2.1.2. Examples

```
RJ   GETNEXTCHARACTER
JP   B6+STATETABLE-1
IF   X1=0 GØTØ ERRØR
X2 = -X0&X2
X1 = LCYCLE EØRBIT-EØIBIT
X4 = UNPACK X4
X7 = X7 + X7
A7 ← -B1 + A7
X6 ← [CHAR+140B]\\200B
```

2.1.2. Semantics

Each machine instruction statement results in generating a 15- or 30-bit machine instruction, possibly preceded by an NØ instruction (in the case of a 30-bit instruction with only 15 bits remaining in the current word.) But first, the assembler assures that the instruction will be aligned on a quarter-word boundary, padding with zero-bits if necessary. The effect is as if the following statement were executed (cf. section 2.2.3.):

DATA [*P\\15]\0

The JP , RJ , PS , XJ , and IF Bi=Bi instructions set the force flag to *true* which will cause the next instruction to be forced upper unless it is labeled by '-' . The instructions PS , RE , WE , and XJ are forced upper automatically unless they are labeled with '-' (cf. section 2.).

The syntax allows several notations for some machine instructions. For example

A6 ← -B1+A6 and

A6 ← A6-B1

are equivalent. Two "new" instructions are provided,

$$X\text{-}reg = \left\{\begin{matrix}\text{RCYCLE}\\ \text{HOLE}\end{matrix}\right\} \textit{expression}$$

They are treated as if they were written:

$$X\text{-}reg = \left\{\begin{matrix}\text{LCYCLE}\\ \text{MASK}\end{matrix}\right\} 60 - [\textit{expression}]$$

Most machine instructions should be self-explanatory, but several deserve some attention. Wherever an option containing a B-register is omitted, the assembler supplies B0 . The IF series generate X register jump instructions and B register comparison-jump instructions. Note that only certain relations between an X register and 0 are allowed. Where a 0 is supplied in a comparison with a ~~B-register~~ B0 is assembled. The statements of the form *X-reg* = [-] *X-reg* [{& | ! | ?}] *X-reg* generate boolean instructions, with the leading '-' signifying complementation of the final operand, and the binary '&', '!', and '?' indicating respectively the AND, inclusive ØR , and exclusive ØR functions. In statements of the form *X-reg* = *X-reg* *arith-op* *X-reg* , arith-op's with a single trailing dot are floating, with two trailing dots are double, with preceding and following dots are rounded.

X-reg = *X-reg* {+ | -} *X-reg* generate integer add and substract instructions.

2.2. Data Generating Statements

2.2.1. Syntax

data-stmt ::= DATA *expr-list*; | {STRING | LINE} *string*;

expr-list ::= {, • *expr* •••}

2.2.2. Examples

```
TENS:DATA 1E0, 1E1, 1E2, 1E3, 10000.0
DATA 1/TRCFLG, N\TYPE, 60-[N+1]\VALUE
ERRMES: LINE 'ERROR IN RULE ABØVE.'
```

2.2.3. Semantics

Statements which generate data words are of two types: those whose operand is an expression (and thus is limited to at most 60 bits in length), and those which take an indefinite-length, string operand.

The DATA statement places a series of expressions into successive machine words, packing short items from left to right wherever possible. Whenever an expression is encountered whose value is "longer" than the amount of space in the current word being assembled, then the offending expression is forced upper. Since only expressions of the form literal ~~expr~~\expr have a length attribute other than 60, the DATA statement usually creates a series of words each containing a single expression.

In order to generate a text coded in internal ASCII, and possibly in System Standard Text format, the STRING and LINE statements may be used. The operand for each is a string. Both generate enough words to hold the string, eight 7-bit characters per word (with the extra 4 bits in each word, normally containing zero-bits in the high-order position), and both pad out unused positions in the last word with blanks (=00) .

However, LINE additionally sets the first 4 bits of the first word to 11B , the SYSTEXT beginning-of-line flag, and inserts a carriage-return character (=155B) at the end of the given string.

2.3. Symbol Definition Statements

2.3.1. Syntax

expr

symbol-asgn-stmt ::= *symbol* {← | =} *expr-list*

2.3.2. Examples

?

change ; to :

TRCFLG = 1

NUMBERØFTEMPS ← NUMBERØFTEMPS + 1

→ XØRMASK = 4\0,2*7\0,7\'.'?';', 2*7\0,7\'.'?':'

HERE: HIER ← *

2.3.3. Semantics

A symbol assignment statement is used to furnish a value which may be referenced when the symbol appears in an expression. When the '←' operator is used to assign a value, the symbol may be reassigned different values (but only with the '←' operator). The symbol keeps only its last assigned value. Permanent definition results when the '=' operator is used. The symbol may never be redefined with either operator.

The expression list which occurs in a symbol assignment is used to create a 60-bit value in much the same mammer as if it were used in a DATA statement. This is, the values of the elements of the list are concatenated together. However, the sum of the length attributes must be ≤ 60 , and if it is less, the value will be padded on the right with zero-bits. The expressions must not contain references to forward-defined symbols, that is symbols which are defined as labels or through '=' assignments *after* the symbols are referenced.

how about [illegible]

2.4. State Changing Statements

2.4.1. Syntax

misc-stmt ::= use-stmt | *positioning-stmt* | *reservation-stmt*

use-stmt ::= USE {*use-block-name* | }

positioning-stmt ::= {ORG | LOC} *expr*

reservation-stmt ::= {BSS | BSSZ} *expr*

2.4.2. Examples

```
ENDØFØLDBLØCK:  USE NEWBLØCK
USE *
ØRG 0
INFILE:  BSS DSCRØLEN+BUFFLEN
```

2.4.3. Semantics

The assembly process consists of two phases. First the source statements are scanned and a skeletal version of the object program is formed consisting of one or more sections called *use-blocks*, which are to be concatenated together at load ~~three~~ time. The order of the sections in the final object program is determined by the source program, but since the length of each is not known until~~l~~ phase two, the values of relocatable symbols in all but the first use-block are not known until phase two.

Thus phase two serves to provide absolute values for symbols which were referenced prior to their definition, albeit partial. This second phase is actually performed by the loader, leaving the assembler proper only the task of producing an ordered sequence of use-blocks, as well as a symbol table and other information for the debugger.

At any moment, the origin, location, and position counters and the force upper flag define the assembler's position and status within the current use-block. The statements described in this section serve to alter, save, and restore these indicators, determine the sequencing in the object program of the use-blocks, and reserve and initialize data areas. Because the assembler uses only one phase, no forward references may appear in the expressions contained in the statements described in this section.

The USE statements with a use-block name as an operand has several effects. The current origin, location, and position counters and force upper flag are set from the values associated with the specified use-block name in the use-block order list (they are 0, 0, 60, and *don't care* initially). The old values are placed in the use-block order list entry for the previous use-block-name, after making a new entry at the end of the list if none existed. Finally an entry is pushed on the use-block push-down list for the old use-block name.

When a USE statement with an '*' operand is executed, the effect is to pop an entry from the use-block push-down list and reinstate the counters and flag from the corresponding use-block order list entry. However, if the push-down list is empty, a USE * statement has no effect.

The positioning statements ØRG and LØC are used to alter the contents of the origin and location counters. Thus LØC $\underset{\sim}{\text{expr}}$ sets the location counter to the value of the expression and forces upper. ØRG $\underset{\sim}{\text{expr}}$ causes the current origin and location counters to be set to the value of the expression, the position counter to be set to 60, and the current use-block name to be set to the use-block designated by the value of the expression (see relocatable symbols). As with USE, the old counters and flag are saved and the old use-block-name is pushed down.

The reservation statement BSS *expr* forces upper and then increments the origin and location counters by the value of the expression, BSSZ *expr* functions similarly except that the reserved area is preset to zeros when the program is loaded.