CGTM 107
C. Riedl
J. Steffani
October 1970

Internal Logic Manual
for the
OCTAVIA Assembler

1. Purpose and environment of the assembler

The assembler accepts one or more programs written in the OCTAVIA assembly language, translates them into MLP-900 machine programs and allocates storage for all data defined in these programs. The generated object programs are at this point absolute, but provisions are made to generate relocatable programs eventually should that become desirable.

The object code is generated in ICAP II binary card word format.

The complete assembler output can be controlled by the user by specifying some options on a // PARM card as explained in the User's Guide and consists of the object program, a source listing including error messages, a cross reference table, and some test information to help debugging.

The assembler is written in PL/1 and currently works under OS on the 360/91. Storage requirements are at present 250K bytes.

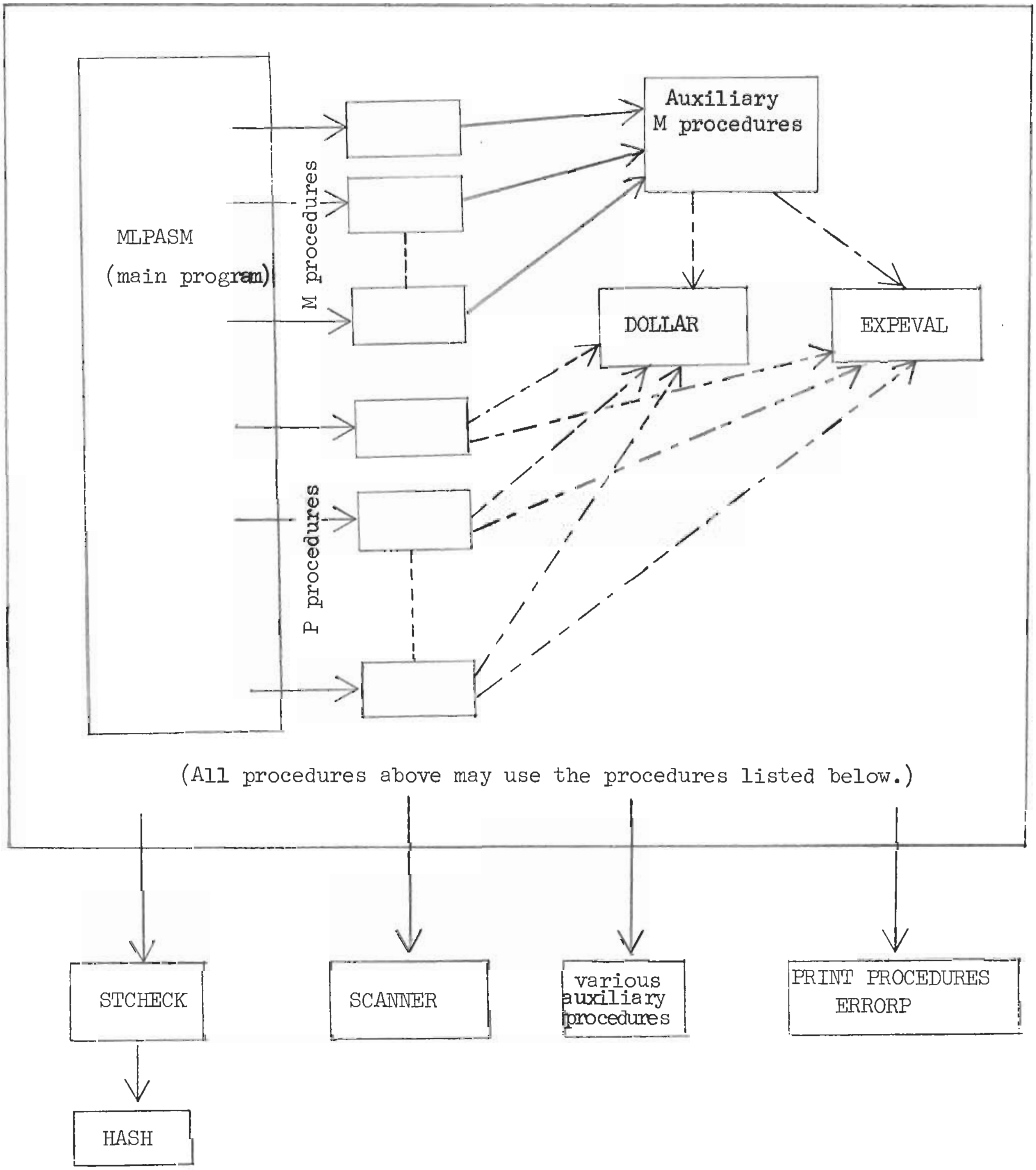

Table 1

General Structure and Functional Flow of the Assembler

2. Organization of the assembler, description of the main program

The assembler operates in one pass and consists of a main procedure MLPASM and various classes of subroutines as shown in Table 1. The functions of MLPASM are explained in this section; the subroutines will be described later.

2.1. Interpretation of assembly options

The user specifies the options for the assembly on a // PARM card in the format described in the User's Guide. The option field is submitted as a parameter to MLPASM.

The following options can be specified:

DECK or NODECK	controls generation of the object code
LIST or NOLIST	controls generation of the source listing
XREF or NOXREF	controls generation of the cross reference table
TEST or NOTEST	controls generation of test information
STSIZ = ... FIXSIZ = ... FIX2SIZ = ... SKSIZ = ...	control the size of the tables used by the assembler (see Appendix A)

In the absence of specified options, or if options have been specified incorrectly, the defaults described the the User's Guide are used.

A listing of the options valid for the assembly is generated.

2.2 Initialization

For each distinct source program, MLPASM allocates storage for all tables (namely the symbol table, the fixup tables and the stack used for expressions see Appendix A).

All table entries and various constants and switches are initialized.

2.3 Main loop to handle one source card

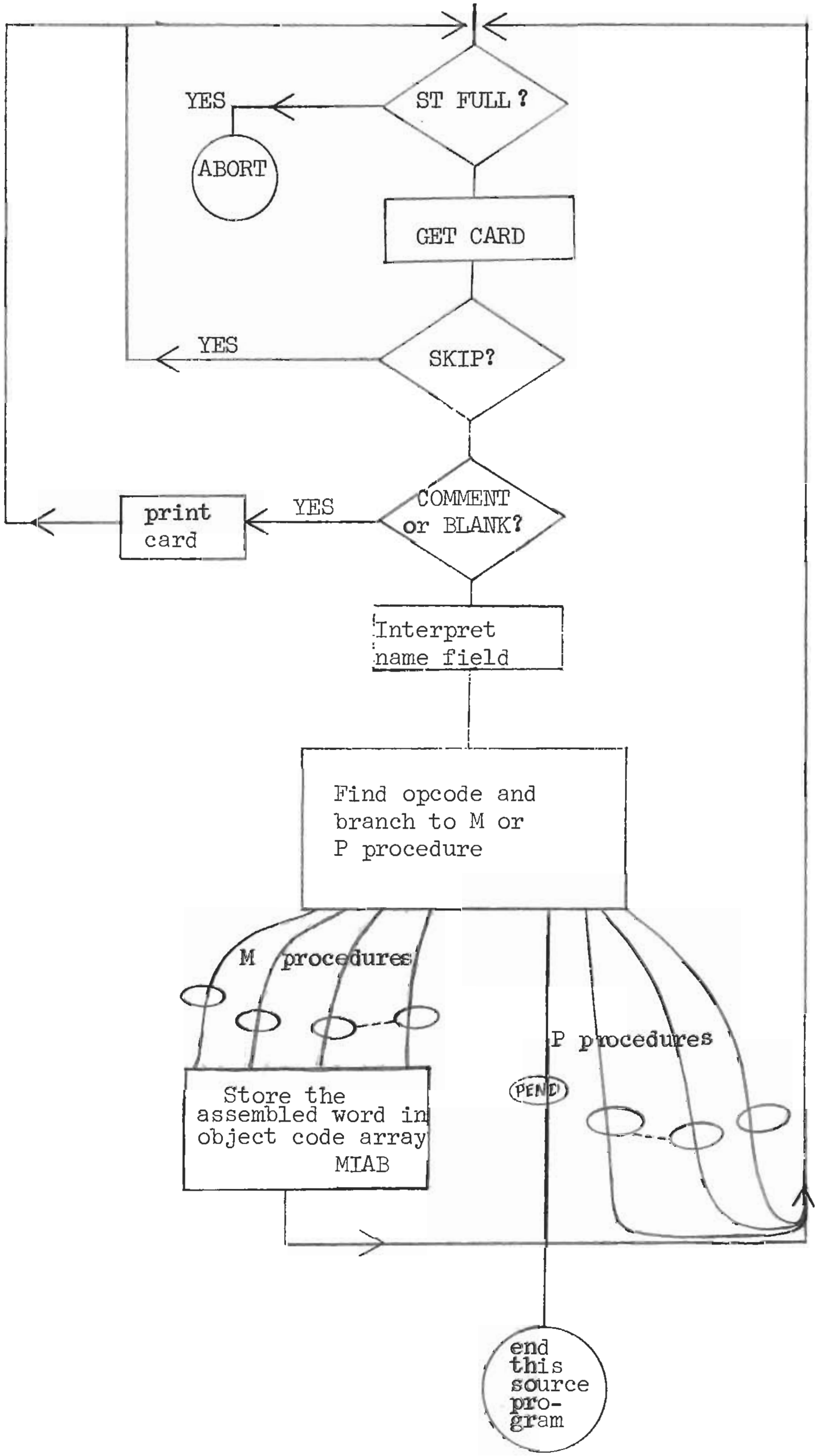

Table 2

Flow Chart of the Main Assembler Loop

Explanation of Table 2:

If an overflow of the symbol table ST has occurred while processing the previous source statement, the assembly is terminated.

Otherwise a new card is read from the main input stream (CARDS) or from a data set specified in a previously encountered COPY assembler instruction.

The SCANNER (Section 6) is called to determine the first field of the statement. The statement can be:

1. Any source statement that is to be skipped according to a SKIP instruction in effect.
2. A comment statement (starting with *) or a blank statement to be included in the source listing and then skipped.
3. A ministep or pseudo instruction statement.

In the third case the scanner returns a name field in FIELD or a Field Length (FL) of zero indicating no name field.

Valid name fields are program point definitions of the form "NH" which are processed by procedure PPOINT (7.4) or identifiers which are entered in the symbol table by STCHECK. A check for multiple definition is made and a message for illegal name fields is issued.

The SCANNER next determines the opcode field which is required for all statements. There is a table, OPCODE, listing all valid codes, and binary search is used to find the index of the current opcode. This index is used to call a particular procedure to process the current statement.

There are two classes of such procedures:

M procedures to interpret ministep statements and to generate a machine word (CMI) corresponding to the current machine instruction.

After completion of an M procedure, control returns to STORE-CMI to store the generated word (sometimes two words) in the array MIAB containing the object code produced.

P procedures to interpret pseudo instructions. After completion a check is made to ensure that the name field did not contain a program point definition.

After finishing processing any statement except "END", control returns to the beginning of the main loop to get the next card.

2.4 Termination of the assembly

Processing an individual source program is terminated by finding an END statement. After completion of PEND (see 4.6), initialization (2.2) for processing the next source program is done, unless an END OF FILE condition indicating the end of the source text is reached.

Symbol table overflow in a source program causes termination of the entire assembly.

3. M procedures and auxiliary M routines

M procedures serve to process the operands of ministep statements. They use a package of auxiliary M routines to build up certain types of operands occurring in more than one ministep. The information on the source statement is coded and stored in CMI ("current machine instruction") which already contains the translated opcode and is to be included in the array MIAB of produced object code upon return from the M procedure.

All M procedures are implemented as entry points of the procedure MPROC. MPROC also contains the auxiliary M routines as subprocedures or as labels (See Appendix C).

3.1 MGEAR

Corresponding opcodes:

GEAR, GEARC, GEARCT, GEART, GEARTC .

The SCANNER (Section 6) determines the first operand indicating the GEAR expression. The format of a GEAR expression is:

[¬ or -] op A binary operator [¬] op B [+C]

where only certain configurations of operators are permitted.

The expression is processed in the following order:

First the optional leading ¬ or - and the binary operator are determined; then OP_A (3.19) is called to evaluate the first operand. The optional ¬ and + C around op B are localized and the operand is evaluated by procedure OP_B (3.19).

An arithmetic combination resulting from the unary and binary operators and the optional '+C' is used to determine the validity of the configuration and the resulting arithmetic code to be stored in CMI. Use MA_SHI to evaluate the remaining GEAR operands.

3.2 MCEDE0

Handles ROW and WSS. They have no operands, so this is a dummy procedure.

3.3 MCEDE1

Handles all CEDE ministeps having op A only: WOP, WOPT, WAS, WAST. SCANNER determines the field of op A; procedure OP_A_EX evaluates the operand.

3.4 MCEDE2

Processes CEDE ministeps with two operands: FIN, FOP, GAD, RMW, SOP, WIF, WIN, WOF, WON, and any of the above followed by "S", "ST", "TS", "T".

SCANNER and OP_A evaluate the first operand, SCANNER and OP_B the second operand. SCANNER determines whether there is any main memory operand.

If it is "CC (Condition Code)", then use V_EXP2 to evaluate the condition code; it is to be stored in CMI bit 10-12. Otherwise give error message.

3.5 MSHIN

Processes all SHIN ministeps. SCANNER and OP_A process the first operand. For Multiply and Divide, SCANNER and OP_B process the second operand. MA_SHI handles the mask and shift operands.

3.6. MCHA

Processes all CHAL/CHAR ministeps. The first operand of these is a CHA expression of the form

[¬ or -] op A binary operator [¬] op B [+C]

where only certain configurations of the operators are permissible.

The expression field is determined by the SCANNER and processed as follows:

First find the leading unary operator, if any, and determine the binary operator.

Then look for byte specification ".0", ".1", ".2", ".3", ".I" at end of op A and insert it into CMI. Call RFD, check whether TYPE = 3 (general register). If it is direct, insert register number into CMI bits 20-24; if indirect, check for "|1" (set bit 24); set bit 19, and insert pointer register into bits 20-23.

Localize possible "¬" before, and possible "+C" after op B.

Evaluate op B: check for "B (expression)" indicating an immediate byte; if found, use V_EXP2 to determine its value and insert it into bits 15,16, 27-32. Set BSEL to one. If not found, look for byte specification. Call RFD, TYPE returned must be 3 (general register) or 5 (pointer register). If general register, insert byte specifications into CMI bits 15,16, or set BSEL=3; if it is specified directly, copy register number into CMI bits 28-32; otherwise set CMI bit 27=1, check for "|1" (set bit 32), copy register number into bits 28-31.

If it is pointer register, make sure there was no byte specification, set BSEL= 2 and copy register number into CMI bits 29-32.

Determine validity of operator configuration and CHAL/CHAR code by looking at table BRCODES, put code in bits 5-8.

Then call SCANNER to look for mask operand. If none there, use default, otherwise call RFD to evaluate field, TYPE must be 4 (mask register), its number goes into bits 9-12 of CMI.

3.7 MGENT

Handles all GENT ministeps. First operand field,determined by SCANNER, must be

op A $\longleftarrow$ op B   or

op A $\longrightarrow$ op B .

Find operator, if it is "$\longrightarrow$", set CMI bit 6=1. Process first operand by calling OP_A_EX.

Call RFD to determine second operand ; TYPE must be 3 (general register), 4 (mask register - set GROUP in bits 25,26 of CMI to one), 13 (miscellaneous register - GROUP = 2), or 14 (Control engine - GROUP = 3). For types 3,4,13 check for indirect specification. If direct, copy register number into bits 28-32 of CMI; otherwise set bit 27; set bit 32 if there was "| 1"and copy register number into bits 28-31.

Then call scanner for parity operand; if it was specified, insert it (0,1,2 or 3) into bits 7,8.

3.8 MBB

Processes BRAT, BENT and BORE ministeps.

The first operand field, determined by SCANNER, must contain a test expression of the form

op A [operator op B]

where the operator can be $\rightarrow$, |, :, & .

It is processed as follows:

First localize the operator, if any. Evaluate the op A by calling TEST_BIT (3.26). If operator was found, insert corresponding test mode into bits 5, 6 of CMI, then process op B by calling TEST_BIT again.

If no operator was found, copy op A field (bits 7, 9-16 of CMI) to op B field (bits 8, 17-24) and set test mode = 1.

Then call REL_ADDR to process the second operand of the ministep.

3.9 MBRAD

Handles all BRAD ministeps.

The SCANNER determines the first operand field, RFD is called, it must return TYPE = 17 (pointer register), and the number of the register is copied to CMI bits 9-12.

V_EXP(3.18) is called to evaluate the second operand (the decrement), the expression returned is truncated to four bits and stored in CMI bits 13-16.

The remaining operands are processed by calling procedures TEST_BIT and REL_ADDR.

3.10 MBC

Processes BCA, BCAE, BCR, BCRE.

The SCANNER determines the first operand field to be processed by TEST_BIT.

A_EXP(3.17) is called to evaluate the second operand, an address expression, which for BCA, BCAE specifies the branch address, but for BCR, BCRE it specifies the relative branch address.

In either case, if the address is undefined, the entries in the fixup table FIX are completed appropriately (see Appendix A: Table formats).

If the instruction is BCA,BCAE, and the expression is defined, its 16 low order bits are stored into CMI bits 17-32. If the address is relocatable the statement is flagged as relocatable by procedure RELOC (a dummy procedure at this point).

If it is BCR, BCRE then the address minus the continuation address is stored into bits 17-32 of CMI. If the original address was not relocatable this is a source of potential error, should we ever produce relocatable code. A warning is given.

3.11 MBI

Processes BIA, BIAE, BIR, BIRE

SCANNER determines the first operand field, RFD is called to process it, the TYPE returned must be 17 (= pointer register), its number is stored in CMI bits 9-12.

For BIA, BIAE there is a second operand indicating the branch address. It is evaluated by A_EXP and treated like the branch address in BCA, BCAE (see 3.10).

3.12 MBLOT

Handles all BLOT ministeps.

The only operand is processed by REL_ADDR (3.24).

3.13. MMAST

Handles MAST ministeps.

They have only one operand field (determined by the SCANNER) consisting of result status bit ← op A operator op B.

First find the arrow and call RFD to evaluate result status bit. Upon return TYPE must be 21 (flip-flop). Copy its number into CMI bits 25-32.

Then locate the operator and put the corresponding logic mode into CMI bits 5,6. Op A and op B can both be processed by calling TEST_BIT.

If no operator was found, evaluate op A with TEST_BIT, then set logic mode to 1 and copy op A field in CMI (bits 7, 9-16) to op B field (bits 8, 17-24).

3.14 MMAR

Processes some of the MOVE ministeps, namely MAR, MCL, MDB.

Their first operand (a CE register designator) is evaluated by CERD (3.23) and stored in CMI bits 17-24.

The second operand,again a CE register, is evaluated by CERD and stored in bits 9-16.

For all ministeps but MDB, procedure IMM (3.25) will process the immediate mask.

3.15 MMSI

Handles MSI only.

Call CERD to evaluate first operand, store it in CMI bits 17-24.

Second operand specifies an 8 bit immediate value, as determined by procedure V_EXP. It is stored into CMI 9-16.

IMM processes the third operand.

3.16 MMOM

Handles MOM only.

First operand (evaluated by CERD) is CE register and is stored in CMI bits 17-24.

Second operand field is determined by SCANNER, interpreted by RFD, which must return TYPE = 21 (for flip-flop). The number of the flip-flop is stored in CMI bits 9-16.

These were all M procedure; now all auxiliary M routines will be explained.

3.17 A_EXP

First SCANNER determines the field of the next ministep operand, which is supposed to specify an A-type expression.

EXPEVAL is called to evaluate the expression. If it returns TYPE = 1 (absolute) a warning is issued that the expression will be truncated.

3.18 V_EXP and V_EXP2

They evalute a ministep operand which is supposed to be an absolute (V-type) expression. The only difference between the two procedures is that V_EXP calls the SCANNER first, whereas V_EXP2 accepts the expression field as a parameter.

EXPEVAL is called to evaluate the expression, TYPE must be 1 (absolute) or an error message is given.

3.19 OP_A and OP_B

Evaluate and insert into CMI the op A (op B) for GEAR and SHIN ministeps and for the CEDE ministeps with two operands.

OP_A: RFD is called, the TYPE returned must be 3 (general register). It may have been specified directly (INDC=0), in which case CMI bits 20-24 are set to the register number.

Otherwise IA bit (= bit 19 of CMI) is set. If there was a "|1" in the specification, bit 24 = 1. Then insert pointer register number into CMI bits 20-23.

OP_B: Operand may be 'S(expression)' or 'L(expression)' indicating immediate operands or a general or pointer register.

If it is immediate operand, call V_EXP2 to evaluate expression, then for short immediate operands set BSEL (= bits 25,26) to two and store the 6 low order bits of expression into bits 27-32 of CMI; for long immediate operands set BSEL = 3, store 4 high order bits of expression in bits 29-32 of CMI and use CMI2 to hold low order bits of expression.

If it is not immediate operand, call RFD. If TYPE returned is 3(general register), then if it was specified directly (INDC=0), insert register number into CMI bits 28-32; if it was specified indirectly, check for '| 1' (set bit 32), then set bit 27 and insert register number into bits 28-31.

For pointer registers set BSEL = 1 and insert register number into bits 28-31.

3.20 MA_SHI

Handles the mask and shift operands for GEAR and SHIN ministeps.

SCANNER determines operand field. If field length (FL) is zero, then both operands are omitted. Put default mask register number into CMI bits 9-12 (error, if none has been specified), shift amount defaults to zero.

Otherwise check for mask operand first: if it starts with $, call DOLLAR (Section 8), it returns TYPE which must be 4 or it is error. Insert register number into bits 9-12. Get next operand field. If SCANNER returns FL non zero, it must be shift amount, set MASK = 1 and go to shift evaluation.

If operand is identifier then call STCHECK(Section 7), if TYPE is 4 interpret it as mask operand as before, otherwise type must be 1(absolute) and it must be shift amount.

Shift evaluation: V_EXP2 evaluates the operand field and must return an V-type expression; if it is negative, bit 13 of CMI is set to one. The absolute value of the expression must be one of 0, 1, 2, 9, 8, 12, 16 and the shift code corresponding to it goes into bits 14-16 of CMI.

If there was no mask operand as yet, (MASK = 0), call SCANNER for next operand field. If there is none, put default into bits 9-12, otherwise call RFD, TYPE must be 4 and the register number is stored in bits 9-12.

3.21 OP_A_EX

Processes the extended op A for WOP, WAS and GENT ministeps. This opernad must directly or indirectly specify an OE register (types between 3 and 13) or the control Engine Data Bus (type = 14).

Call RFD, check whether type is correct, then insert into CMI bits 13-16 the corresponding group code. If TYPE = 14, this is all.

Otherwise check for indirect specification. If it is direct, then for types 5 through 12 insert first 3 bits of its number into bits 10-12. Bits 20-24 receive the five low order bits of the register number.

If it is specified indirectly, put pointer register number into bits 20-23, set bit 15 and if there was ' | 1' set bit 24.

3.22 RFD

This procedure evaluates a register or flip-flop designator, i.e. a predefined RF constant starting with $ or an identifier previously equated to an RF constant (its type in ST must be > 2).

Parameters of RFD are the field and field length of the desired thing.

If there are no characters, report error.

If it starts with $ call DOLLAR.

Otherwise it must be identifier. Call STCHECK and check whether type is correct.

Both DOLLAR and STCHECK return TYPE and VALUE (containing register number).

3.23 CERD

Finds an operand that specifies a CE register by calling RFD and making sure that TYPE is between 15 and 20.

3.24 REL_ADDR

Computes 8 bit relative address and inserts it into CMI.

A_EXP evaluates address.

If it contains forward references (TYPE = 0), entries in the FIXUP table FIX are completed (KIND = 3), otherwise the difference between the address and the continuation address is computed. If this difference is between -128 and +127 it is stored in bits 25-32 of CMI, otherwise an error message is given.

3.25 IMM

Finds an 8 bit immediate mask.

SCANNER determines the operand field, if FL = 0 then the default (all one's) is stored into CMI bits 25-32.

Otherwise V_EXP2 evalutes the operand, which is then stored into bits 25-32.

3.26 TEST_BIT

Evaluates an operand specifying a test bit, i.e. a flip-flop designator possibly preceded by "¬".

This procedure has four parameters: the operand field, its length, the receiving field for the test mode (= a bit to be set if there was no "¬"), and the receiving field for the flip-flop number.

Localize "¬" first, if absent, set appropriate bit. Then call RFD to evaluate flip flop. Upon return, TYPE must be 21. Insert flip flop number into the receiving field specified by the fourth parameter.

4. P procedures and auxiliary routines

All P procedures and auxiliary routines described in this section are entry points in procedure PPPPP (See Appendix C). The P procedures not contained in PPPPP are described in Section 5.

4.1 PMASK - - - entry point

Processes the MASK assembler instruction.

SCANNER evaluates the operand field which must contain '=NONE' indicating no default mask register or it must contain an identifier (processed by STCHECK) or an RF constant (processed by DOLLAR) both specifying a mask register (TYPE=4). The result is indicated in a global variable.

4.2 PALIGN - - - entry point

Processes the assembler instruction ALIGN.

The operand field, determined by SCANNER, must contain an absolute expression which is processed by EXPEVAL. Its value must be > 0 and the location counter is adjusted to a multiple of the value. If the instruction had a name field, then its entry in ST is assigned the resulting value of the location counter.

4.3 PBSS - - - entry point

Processes the assembler instruction BSS.

The operand field is evaluated as in PALIGN. If the instruction had a name field then the identifier will have the value of the current location counter. The location counter then is incremented by the value of the operand field.

4.4 PBES

Processes the assembler instruction BES.

Identical to PBSS except that the identifier in the name field, if any, gets assigned the resulting value of the location counter-1.

4.5 Entry Point PCOMSY:

The assembler instruction COMSY is processed by this routine.

If there was a name field, STREMOV is called to ignore it.

All operands (determined by SCANNER) must be identifiers. For each of them, STCHECK establishes a section-bound entry in ST. Then a common symbol is formed by prefixing the identifier with section number 'FF'; it is entered in ST, if not found there. The common pointer STCOMP of the section-bound entry is used as a link to the corresponding common entry and the value of the identifier, if any, is copied to the common entry or vice versa.

4.6 Entry Point PEND:

The assembler instruction END is processed by this entry. The END card may contain Miniflow Status Word initialization information. If so, the MSW is set to the value resulting from this evaluation. If the LIST option is in effect, a call is made to the PRINTL routine to print the assembly listing. The binary deck is next produced. Storage allocated is freed to provide room for the SORT routine XSORT to be brought into core. The SORT routine is called only if the XREF option is selected. Then control is returned to the main driver.

4.7 Entry Point PEQU:

The assembler instruction EQU is processed by this entry. A test is made for EQU * and if found, the identifier is given the value of the location counter. Otherwise the operand is evaluated by EXPEVAL and the result is stored in the STVALUE field of the name field.

4.8 Entry Point PCOPY:

The assembler instruction COPY is processed by this entry. The operand is DDname which will subsequently be opened. Switches are set to insure this and an exit is made.

4.9 Entry Point PDATA:

The assembler instruction DATA is processed by this entry. The operand field(s) are evaluated and the corresponding machine words are initialized to contain the values found.

4.10 Entry Point PINITR:

The assembler instruction PINITR is processed by this entry. The operand fields are evaluated and binary cards are produced to initialize the various registers indicated.

4.11 Entry Point PØBJEND:

This entry point punches a loader control card to indicate end of binary deck.

4.12 Entry Point POBJCON:

This entry point punches a loader control card to indicate end of load module.

4.13 Entry Point CKSUMM:

This entry point calculates the end around carry checksum for the binary card.

4.14 Entry Point PUNCOLB:

This entry punches the column binary card if the DECK option has selected.

5. Printing Procedure PPPPPP

This general purpose procedure contains seventeen (17) distinct entry points. The action resulting from each will be detailed below. The procedure could perhaps be best described as that which handles the printing and formating. A list of the entry points and the routines they call is shown in Appendix C. The following P procedures are entry points in this procedure: PSPACE, PTITLE,PPRINT,PEJECT,PSKIP,PSECT,PORG .

5.1 Entry Point INITPP:

This entry initializes or presets various switches used in the remainder of the procedure. These include first time switches to provide for opening the data sets associated with the temporary print, the cross reference table, and errors discovered at fixup time. The title is preset to blanks as well as the section name table.

5.2 Entry Point PRINTR:

This entry is used to indicate that a text record is to be placed on the intermediate print data set. The format of entries is shown in Table 3. The print code associated with the entry is 1.

FORMAT OF ENTRIES IN THE TEMPORARY PRINT DATA SET

Description	Type	Length
Pointer to stored machine instruction	integer	4 bytes
80 column card image	character	80 bytes
Type of print record	integer	4 bytes
Location counter value	integer	4 bytes
Card number	integer	4 bytes

Table 3

5.3 Entry Point PSPACE:

The assembler instruction SPACE is processed by this entry. An intermediate print record with a print code of 5 and the number of lines to space contained in the pointer to the stored machine instruction is placed on the print data set.

5.4 Entry Point PTITLE:

The assembler instruction TITLE is processed by this entry. An intermediate print record with a print code of 3 and the title text is placed on the intermediate print data set.

5.5 Entry Point PPRINT:

This entry processes the assembler instruction PRINT ON/OFF. An intermediate print record is made containing the print code 6 for ON and 7 for OFF.

5.6 Entry Point PEJECT:

This entry processes the assembler instruction PEJECT. An intermediate print record is made containing the print code 4.

5.7 Entry Point ERREND:

This entry processes the error records produced at fixup time. A record containing the error text and code and statement number is placed on an intermediate data set.

5.8 Entry Point ERRORP:

This entry processes the error messages produced by the assembler. The error messages along with a print code of 2 are placed on the intermediate print data set.

5.9 Entry Point PCOMNT:

This entry processes the comment card entries. An intermediate print record is written with a print code of 8.

5.10 Entry Point PLOCMI:

This entry processes print records which are to contain only the location and the machine instruction - no text or statement number. An intermediate record is written with the above information and a print code of 9.

5.11 Entry Point PCMIØN:

This entry processes print records which are to contain the machine instruction only. An intermediate print record with a code of 10 is produced.

5.12 Entry Point POLAV:

This entry processes print records which are to contain the location and value fields only. An intermediate print record is produced with a code of 11.

5.13 Entry Point PRINTL:

This entry is used to produce the assembler listing. The date and time of day are obtained and the intermediate print file is repositioned and processed. Those records which contain printer command instructions are effected. Error messages are held temporarily in order to insure their position following the source record to which they apply.

5.14 Entry Point XREFR:

This entry processes the cross reference entries. Records are created for each incidence of a symbol and are stored on a temporary data set. The format of this data set is shown in Table 4.

FORMAT OF ENTRIES IN THE TEMPORARY CROSS REFERENCE FILE

Description	Type	Length
section name	character	8
section number	character	2
SYMBOL	character	8
statement number	integer	4

Table 4

5.15 Entry Point PSKIP:

The assembler instruction SKIP is processed by this entry point. The expression is evaluated. For the SKIP to become effective, the value of the expression must be odd if it is SKIPT; for SKIPF the value must be even. If this holds, the second operand indicating a name field is stored in an external character string for processing by the main driver, where the source statements are read. Also a global indicator for SKIP in effect is set.

5.16 Entry Point PSECT:

The assembler instruction SECT is processed by this entry. A limit of 20 distinct sections with section names of 8 or less characters is allowed. The section numbers are assigned at this entry and consist of a 2 character string containing the section number. These are prefixed to the local symbols in a section thus producing unique identifiers on a section basis.

5.17 Entry Point PORG:

The assembler instruction ORG is processed by this entry. The ORG value field is tested for validity and range. Also a high water and low water origin value is kept.

5.18 Sub procedure BITHEX:

This procedure converts a 32 bit value into its eight character hexadecimal representation for print purposes. The location and machine instruction fields are examples of data converted to hex by this procedure.

6. The SCANNER

This is a routine that determines one field of an instruction (i.e. the name field, op code field or an operand field). These fields must be contained in column 1-72 of one card.

It maintains two pointers, one of which always points to the first character of the field to be processed, and TEXTPO which points to the current character to be processed.

SCANNER returns FIELD containing the characters of the instruction field and FL containing the field length. If FL=0 then no field was found; otherwise FL is computed by subtracting the two pointers.

Upon each entry FIELD is preset to blanks.

If the SCANNER is looking for the name field, which must start in column one, it may return

FIELD = '*', FL=1 for a comment card

FL=0 for no name field

or FIELD will consist of all characters up to the first blank. Both pointers are then set to point to the first following non blank character.

If the opcode field is desired, all characters up to the first blank are collected in FIELD. The pointers are set to indicate the first following non blank character. Then FIELD and FL are returned.

If an operand field must be determined, all characters are collected up to the first comma or blank. The pointers are positioned on the first character after comma. If last character was blank then an indicator is set that the following field is the comment field. Then FIELD and FL are returned.

If an operand field is wanted but the next field is already the comment field, then FL=0 is returned. It must be an omitted operand or an error.

7. Symbol table routines

These routines look up entries in the symbol table ST or, if necessary, make or delete an entry in the symbol table.

The symbol table provides a means of rapidly obtaining the attributes of a given symbol (i.e. identifier or program point).

There is an entry in ST for each identifier and for each program point definition. These entries consist of several fields as described in Appendix A which should be read in parallel with this section.

In order to access an entry one must first construct a HASH entry address. This HASH address is the pointer to the symbol table entry which either contains all information corresponding to this symbol, or the information may be obtained by following down a chain of pointers.

7.1 STCHECK:

This is a procedure with two parameters (field and field length) which is used to search for an entry in ST and make an entry if no entry is found. STCHECK returns the index of the ST entry obtained as well as TYPE and VALUE.

The STCHECK routine verifies that the field contains up to 8 valid alphanumeric characters and begins with an alpha character (this check is omitted for program points and for already verified identifiers). The section number is prefixed to the identifier and the HASH procedure called to produce a hash code corresponding to this symbol. This hash code serves as the index to the symbol table for this identifier. The STID field corresponding to this index is tested to see if there has been an entry made at this hash location. If none is found, then the symbol is stored in STID at the location specified by the hash value, the STVALUE field associated with the entry as well as TYPE and VALUE are set to zero, and a return to the caller is made.

If the STID field is not blank then a test is made to see if it matches the current symbol. If a match occurs, then TYPE=STTYPE and VALUE=STVALUE are returned. If no match is made then STCHAIN, the chain value, is tested for non-zero. If a non-zero value is found, then this becomes the index to the symbol table and we continue searching for the symbol in the STID field as before. If the chain value is zero then we have reached the end of the chain without either finding an entry for the present symbol or creating a new entry since we have not found a blank STID. At this point an empty location in the symbol table is looked for. Upon finding an empty location the chain values are established, symbol stored in STID and the index returned. In the event of a symbol table full condition, a global flag is set for testing by the main driver and a return to the caller with index zero is made.

7.2 Construction of the HASH address

Symbols are required to consist of eight (8) or less characters. The section number, range 01 through 19, is prefixed to the symbol for the purpose of creating the HASH code. The HASH routine reverses the order of the characters in the symbol, treating the symbol as though it were ten (10) full characters (2 character section prefixes plus 8 character symbol names). Thus the first character becomes the tenth, the second becomes the seventh, and so on until the first blank is encountered or all ten characters (non blank) have been reversed. As soon as the first blank character is encountered it is replaced by the character A. Each remaining blank character is likewise replaced with the next higher collating sequence character, e.g. the symbol '01FWAbbbbb' in the first section (block) becomes 'EDCBAAWF10'. The resulting ten character string is now summed by dividing the string into five substrings, each containing two characters, and these five substrings are treated as 32 bit integers and added together.

Thus we have:

Character String	Hex Value
ED	0000C5C4
CB	0000C3C2
AA	0000C1C1
WF	0000E6C6
IO	0000F0F1
a value of	000423FD

Next the value resulting from extracting the low order fifteen bits of this resultant is squared. This produces:

$$(23FD)^2 = 050F2809$$

Next, the center bits, 5-28, are extracted. This produces:

0050F280

Next, this value is divided by the symbol table size as specified by the user (or the default value,if not specified). For this example a symbol table size of 600 (decimal) is used. Thus,

0050F280/258 = 2289 + 168 remainder

The remainder 168 (360 decimal) becomes the HASH code.

7.3 STREMOV and STREMNP

These routines serve to delete an entry from the symbol table which is necessary if a name field has been specified in a pseudo instruction which does not allow name fields. STREMOV first gives an error message while STREMNP only deletes the entry.

Deletion of the entry consists in setting all its fields to zero and in resetting the chain pointers so that the access mechanism will work for the remaining entries.

7.4 PPOINT

This routine makes symbol table entries for program point definitions. It is called when 'NH' is found in a name field. It makes up the symbol corresponding to the program point consisting of the section number, the digit (0,1..,9 specifying 'N') and a current number which is incremented for each redefinition of 'NH'. Then STCHECK is called. Upon return the fields are set as follows: STTYPE=2, STTVALUE= location counter.

8. Evaluation of RF constants (DOLLAR routine)

DOLLAR has two parameters: field and length of the RF constant. RF constants can be of 3 forms:

(i) a direct specification of a register or flip flop as in \$P07, \$COP;

(ii) an indirect specification of an OE register, e.g. \$M(\$G01) or \$G(POINTER) where a pointer register must be specified inside the parentheses;

(iii) an indirect specification as in (ii) only an odd number of the OE register is forced as in \$M(\$G02|1) or \$M(POINTER|1).

DOLLAR returns:

TYPE = the type code corresponding to the specified register or flip flop, even if indirectly specified (see Appendix B).

VALUE = the register or flip flop number in a convenient format. These formats are not discussed here.

INDC = 0, 1, 2 if the RFD constant was of form (i), (ii), (iii).

First the possible '(' is looked for. If none is found, i.e. it is a direct specification, the numeric part at the end of the RF constant is split from the rest and the non numeric part is searched for in a table of valid Dollar codes. The resulting index is used to access the array DOLMMV. DOLMMV contains for each dollar code: the corresponding type, the allowable range for the numeric part, and a value to subtract from the numeric part to obtain the true register or flip flop number.

Set TYPE from here, check numeric part, compute value, make up appropriate format depending on type, return TYPE, VALUE, INDC=0.

If it is indirect, look first for '|1' and set INDC=1 or 2 according to whether it is there or not. Inside parentheses there must be another RF constant (check for \$ and treat it as a direct specification) or identifier (call STCHECK); in both cases the result must be a pointer register.

Return in TYPE the type of the indirectly specified register and in VALUE the pointer register number.

9. Expression Evaluation

9.1 EXPEVAL

This is the procedure used to evaluate expressions. It has three parameters:

the expression field,

the field length,

an indicator specifying whether or not forward references are permitted (they are permitted only in expressions specifying a branch address in ministeps).

EXPEVAL uses a stack STK, each entry in the stack has three fields:

STK.OPR	containing the binary operator ,
STK.VAL	containing the 32 bit value of its left operand (which may be an already computed intermediary result),
STK.ACT	containing the A valence count of the left operand (zero if it is absolute, one if it is relocatable, 2 if it is the sum of two relocatable items etc.; the evaluated expression must have ACT zero or one).

Expressions have the general form

[+ or -] term operator term . . . .

and are processed from left to right.

9.1.1 Initialization

Consists of setting the initial stack entry as well as a few indicators to zero.

Then the unary operator, if any, is searched out. If it is ' - ' , an entry in the stack is made with OPR = unary minus, VAL and ACT = 0.

9.1.2 Term Evaluation

Before finding the term which must follow, check for stack overflow and end of text.

Then branch to term evaluation routine corresponding to first character of term. All evaluated terms and their A valence are stacked, the values are also stored in a 36 bit field OPD which is returned if the expression contained no operator (avoiding the truncation occurring otherwise).

The following are the term evaluation routines:

TERM(0): first character was not alphanumeric. If it is * it is a location counter reference; otherwise it is a sequence error and expression evaluation is terminated.

TERM(1): first character is A, could be identifier (if second character is not '(' go to TERM (4)) or A-type constant.

The latter is processed by going through the term evaluation once more, but with indicator ADCON set so that after evaluating the term control will return here. Then check whether the term found inside the A-type constant was absolute and change STK.ACT from zero to one. Look for closing parenthesis at the end of the A-type constant and reset indicator ADCON.

TERM(2): first character is B. Can be identifier, binary self defining term B 'digits' or BC character self defining term BC'character string'. If first 3 characters are not BC' then procedure Box (9.15) is called which checks if it is binary self defining term, and if not, submits term as identifier (TERM(4)).

Upon return from box, the binary self defining term must be assembled bit by bit from the source text into OPD.

If first 3 characters are BC' then what follows should be a string of up to four characters followed by '. Copy source text up to 4 characters into temporary field counting two apostrophes as one character and stopping when a single ' or the end of text is reached. If no ' is found after 4 characters, keep looking for it until end of text is reached, but terminate expression evaluation afterwards. Otherwise copy temporary field into OPD, and its 32 low order bits into stack.

TERM(3): first character is C: could be C'character string' or identifier. If second character is ' interpret it like BC constant (TERM(2)) otherwise it must be identifier (TERM(4)).

TERM(4): Term started with one of the following letters: D to N, P to W, Y, Z or it started with another letter, and it is already clear that it must be an identifier.

Find all characters of identifier. If a valid identifier (up to 8 alpha-numeric characters) results, call STCHECK which must return TYPE < 3.

If TYPE = 1 (2) then term is absolute (relocatable) and there is no problem.

If TYPE = 0 then this is a forward reference. Check whether forward references are permitted (parameter of EXPEVAL) and make sure that preceding operator, if any, was additive, and set indicator UNDEF to be able to check operator after undefined identifier, if any. Set STK.VAL and STK.ACT to zero.

If this is the first undefined identifier in this expression, make an entry in fixup table FIX for the expression.

Then make the entry in FIX2 for the identifier (See Appendix A). Call procedure FIXUP to handle overflow.

TERM(15): first character is 0.

Procedure BOX will determine whether it is octal self defining term or identifier. If it is self defining term the 3 low order bits of each source digits make a 3 bit octal digit.

TERM(24): Same as TERM(15) except that self defining terms are hexadecimal and each source character must be translated to its 4 bit hexadecimal equivalent.

TERM(28)-TERM(37): first character was digit.

Check whether it is program point reference (must be followed by B or F).

If not, assemble decimal constant, check whether it is not too large and convert it.

If it is 'NF' then call STCHECK to make an entry for the next definition of this program point (see table formats Appendix A), then treat it like undefined identifier (TERM(4)).

If it is 'NB' call STCHECK to look it up, give error message if no entry exists, treat it like identifier with TYPE=2 otherwise.

9.1.3 Operators

After a term is found and stacked, look for operator or end of text.

Stack the operator number (end of text has one too).

If precedence number of present operator is not lower than of last one in stack, go to get next term.

Otherwise perform all operations until an operator with a higher precedence number is reached and reduce stack accordingly. For all but + and - check whether type of the operands is absolute.

If the stack does not get emptied this way, then the operator was not the end of the expression, go to get next term.

9.1.4 End of expression

If expression is V-type (STK.ACT(1)=0) then return 36 bit value which is taken from the 36 bit OPD if there was no operator, or from the 32 bit STK.VAL(1) whose sign bit is extended to the left. TYPE = 1.

If it is A-type then the value is the 16 low order bits of STK.VAL(1). TYPE = 2.

If it is undefined the temporary A valence and expression result must be stored in fixup table FIX.

9.1.5 BOX

This is a procedure to evaluate Binary, Octal and heXadecimal self defining terms. First character was B, O, X.

Parameters of box are the allowable digits of the self defining term and the allowable number of digits.

Box first checks for an ' following the first character. If there is none, the term must be an identifier (see TERM(4)).

Then it collects digits, stopping if ', end of text, or invalid character is reached.

If number of digits is wrong or no ' found, an error message is given, otherwise the length is returned to the term evaluator.

9.2 FIXUP

This routine is used to fix up expressions containing forward references. It uses the tables FIX and FIX2 whose format is described in Appendix A. FIXUP can be called for three reasons (specified by its parameter):

overflow of FIX or FIX2 when an entry for a new expression should be made,

overflow of FIX2 while continuing processing an expression,

end of source program.

Go through all entries of FIX (they correspond to distinct expressions), and for each one check through all corresponding entries of FIX2 (accessed by FIX.PTR, their number is FIX.#FIX2).

Look up the identifier (FIX2.STIND) in symbol table; if it is already defined, add or subtract it to the temporary result in FIX.RES and compute FIX.ACT. Then set FIX2.STIND to zero.

If it was possible to evaluate the expression completely then check type of its result. Depending on FIX.KIND make of it:

(1) a 16 bit branch address to be stored in bits 17-32 of the instruction word indicated by FIX.CMI

(2) a 16 bit relative branch address (= value minus continuation address FIX.CONT) to be stored in bits 17-32.

(3) an 8 bit relative branch address (check size!) to be stored in bits 25-32.

After having gone through all entries of FIX that way, reduce tables by first eliminating all FIX entries belonging to completely resolved expressions and by grouping the remaining ones at the beginning of FIX, then by treating FIX2 the same way and resetting FIX.PTR and FIX.#FIX2 while doing it.

Then if it is end of program and if TEST is specified, print unresolved references and return.

If FIXUP is called during expression evaluation, now make the entry that caused the overflow, reset all pointers and return.

If FIXUP was not able to reduce tables sufficiently to be able to make the entry, then give error message, clear tables and make current entry be the first entry. Then return.

APPENDIX A Table Formats

1. The format of the symbol table entries.

There is an entry in the symbol table ST for each identifier and each program point definition (see section 7).

Each entry consists of the following fields:

STID CHARACTER (10)

which contains the symbol, i.e. for identifiers the section number and the identifier extended with blanks to the right; for common identifiers the same, but section number is 'FF'; for program point definitions see 7.4

STTYPE FIXED BINARY (15,0)

which contains the numeric type code as explained in Appendix B.

STVALUE BIT (36)

which contains the value, i.e. if TYPE=1 a 36 bit value
if TYPE=2 a 16 bit address
if TYPE>2 a register or flip flop number in the format used to store it in the machine words.

STCOMP FIXED BINARY (15,0)

This is zero if the identifier has not been declared common and contains the index of the common entry otherwise.

STCHAIN FIXED BINARY (15,0)

This is used to link together different entries corresponding to the same hash value. Contains the index of the next such entry or zero if there is none.

The default size of the symbol table is 2048 entries.

2. The fixup tables

There are 2 fixup tables, FIX and FIX2. There is one entry in FIX for each expression containing forward references and an entry in FIX2 for each forward reference found.

The FIX entries are structured as follows:

FIX.RES BINARY FIXED (31,0)

contains the temporary result of the expression

All other fields in FIX and FIX2 are BINARY FIXED(15,0).

FIX.ACT	contains the A valence count of the temporary result
FIX.PTR	contains the index to the first entry in FIX2 corresponding to this expression
FIX.#FIX2	number of FIX2 entries corresponding to this expression
FIX.#SMTN	contains the current statement number, used for error messages
FIX.CMI	index of the current machine instruction in the array MIAB of generated object code
FIX.KIND	indicates what has to be done with the fixed up expression - can be 1, 2, 3 (see 9.2).
FIX.CONT	contains the continuation address (relevant for KIND=2,3)

The format of each FIX2 entry is as follows:

FIX2.STIND	contains the symbol table index of the entry belonging to the forward reference
FIX2.OPR	contains the information whether the undefined value has to be added to or subtracted from the temporary result.

The default size for FIX is 50; for FIX2 it is 100.

3. The STACK

The stack is used for expression evaluation only and the meaning of its entries is explained in 9.1.

Their format is as follows:

STK.OPR BINARY FIXED (15,0)

STK.VAL BINARY FIXED (31,0)

STK.ACT BINARY FIXED (15,0)

The default size for the stack is 20.

APPENDIX B Numeric Type Codes

The following table gives the type numbers which are used throughout the assembler to indicate what an identifier or an RF constant stands for. The type number of an entity is generally passed in the global TYPE. Types of identifiers are kept in the STTYPE field of the symbol table entry (See Appendix A), and types of RF constants can be found in array DOLMMV (see Section 8).

TYPE Code	Meaning
0	undefined
1	absolute (V-type)
2	relocatable (A-type)
3	general register
4	mask register
5	auxiliary register Bank 0
6	" " Bank 1
7	" " Bank 2
8	" " Bank 3
9	OE language board register Bank 0
10	" " " " Bank 1
11	" " " " Bank 2
12	" " " " Bank 3
13	Miscellaneous registers
14	Control engine data bus
15	Group 0 - state flip flops
16	Group 1 - state flip flops
17	Pointer registers
18	Miscellaneous CE registers
19	Group 4 - subroutine stack
20	Group 5 - subroutine stack
21	State flip flop constants.

Note that directly and indirectly specified registers have the same type.

See the Octavia Manual APPENDIX I for a complete list of all valid RF constants. Their types are the types of the register or flip flop group they belong to.

APPENDIX C Physical Organization of the Assembler

The assembler consists of nine external procedures:

MLPASM ...	the main program
MPROC ...	M procedures and auxiliary M procedures
PPPPP ...	some P procedures and auxiliary procedures
PPPPPP ...	remaining P procedures, printing procedures, etc.
EXPEVAL ..	EXPEVAL and FIXUP
STCHECK ..	STCHECK, HASH, SCANNER
DOLLAR ...	DOLLAR routine
RELOC ...	a dummy procedure at this point
XSORT ...	the sorting routine; it does not interact with the rest

MLPASM consists of the main program only and makes calls on almost all other procedures (except DOLLAR and RELOC).

The following are tables describing the entry points of the other procedures and calls from them.

Organization of the M procedures

PROCEDURE MPROC

ENTRY	CALLS within MPROC	External calls
MGEAR	OP_A,OP_B	SCANNER,ERRORP
MCEDEO		
MCEDE1	OP_A_EX	SCANNER
MCEDE2	OP_A,OP_B,V_EXP2	SCANNER,ERRORP
MSHIN	OP_A,OP_B	SCANNER
MCHA	RFD,V_EXP2	SCANNER,ERRORP
MGENT	OP_A_EX,RFD	SCANNER,ERRORP
MBBB	TEST_BIT	SCANNER,ERRORP
MBRAD	RFD,V_EXP,TEST_BIT	SCANNER,ERRORP
MBC	TEST_BIT,A_EXP	SCANNER,RELOC,ERRORP
MBI	RFD,A_EXP	SCANNER,ERRORP,RELOC
MBLOT		
MMAST	RFD,TEST_BIT	SCANNER,ERRORP
MMAR	CERD	
MMSI	CERD,V_EXP	
MMOM	CERD,RFD	SCANNER,ERRORP

So all M procedures are entry points of MPROC. The auxiliary procedures follow.

Name	Realized as	Calls within MPROC	External calls
CERD	PROCEDURE	RFD	SCANNER,ERRORP
REL_ADDR	LABEL	A_EXP	ERRORP
IMM	LABEL	V_EXP2	SCANNER
A_EXP	PROCEDURE		SCANNER,EXPEVAL
V_EXP	PROCEDURE		SCANNER,EXPEVAL,ERRORP
V_EXP2	PROCEDURE		EXPEVAL,ERRORP
OP_A	PROCEDURE	RFD	ERRORP
OP_B	PROCEDURE	V_EXP2,RFD	ERRORP
OP_A_EX	PROCEDURE	RFD	ERRORP
RFD	PROCEDURE		DOLLAR,STCHECK,ERRORP
TEST_BIT	PROCEDURE	RFD	ERRORP
MA_SHI	LABEL	V_EXP2,RFD	SCANNER,ERRORP, DOLLAR,STFIND

ENTRIES AND ROUTINES CALLED

Procedure PPPPP:

ENTRY	EXTERNAL ROUTINES CALLED
PMASK	PCOMT,STREMOV,SCANNER,DOLLAR,STCHECK,ERRORP
PPOINT	ERRORP,STFIND
PALIGN	SCANNER,EXPEVAL,PLOCMI,ERRORP
PBES	" " " "
PBSS	" " " "
PCOMSY	PCOMNT,STREMOV,SCANNER,STCHECK,ERRORP
PEND	SCANNER,EXPEVAL,ERRORP,PLOCMI,FIXUP,PRINTL,CKSUMM,XREFEND
PEQU	SCANNER,DOLLAR,EXPEVAL,PCMION,PCOMNT,ERRORP
PCOPY	PCOMNT,STREMOV,SCANNER,ERRORP
PDATA	SCANNER,EXPEVAL,PRINTR,POLAV
PINITR	PCOMNT,SCANNER,DOLLAR,STCHECK,ERRORP,EXPEVAL,CKSUMM
POBJEND	PUNCOLB
POBJCON	PUNCOLB
CKSUMM	PUNCOLB
PUNCOLB	

Table 4.1

ENTRIES AND ROUTINES CALLED

PROCEDURE PPPPPP

ENTRY	EXTERNAL ROUTINES CALLED
INITPP	
PRINTR	
PSPACE	STREMNP, SCANNER, EXPEVAL
PTITLE	STREMNP
PPRINT	STREMOV, SCANNER
PEJECT	STREMNP
ERREND	
ERRORP	
PCOMNT	
PLOCMI	
PCMION	
POLAV	
PRINTL	BITHEX
XREFR	
XREFEND	XSORT
PSKIP	PCOMNT, STREMNP, SCANNER, EXPEVAL
PSECT	PCOMNT, STREMNP, SCANNER
PORG	SCANNER, EXPEVAL
BITHEX	

ORGANIZATION OF EXPEVAL

ENTRY	CALLS WITHIN EXPEVAL	EXTERNAL CALLS
EXPEVAL	BOX,FIXUP	ERRORP,STFIND
FIXUP		ERRORP,ERREND

ORGANIZATION OF STCHECK

ENTRY	CALLS WITHIN STCHECK	EXTERNAL CALLS
STCHECK	HASH	XREFR,ERRORP
SCANNER		
STREMOV		ERRORP
STREMNP		
STPRINT		

ORGANIZATION OF DOLLAR

ENTRY	EXTERNAL CALLS
DOLLAR	XREFR,ERRORP,STCHECK