Overview of the LOLO Assembler and Functional Description

Charley Shapiro
May 21, 1974

Technical Memo

TM.74-18

Overview of the LOLO Assembler

The assembler converts input (in the high level language Micro) to binary output; it also generates files for expanded listings, error listings, and identifier / keyword cross-reference listings.

The assembler consists of four modules -- the preprocessor, the parser, the code generator, and the assembler-driver. In addition, there is an unpreprocessor.

The preprocessor converts input strings (in Micro) to strings of one-word tokens. One token is generated for each identifier / keyword/ constant in the input line. In the case where the input line is a comment, the output consists of a comment token followed by the text of the comment in packed form -- 3 characters to a word. The output of the preprocessor, known as preprocessed text, is stored in the data structure PP'TEXT'TABLE.

The Parser takes the preprocessed text (PREP'OUTPUT) as input (INPUT'STREAM) and converts it to token strings of a differing form. Arithmetic statements are changed to reverse polish form. The legality of Boolean expressions is checked and the LB and RB fields of the microword are set. Macros are expanded, new identifiers are stored in the symbol table. Checks are made for syntactic errors and sub-expressions are tagged as being one of five types: assignment statements, branch statements, special functions, field assignments, or memory operations. The output

of the parser consists of two-word tokens; the first word contains type and value information, the second is a pointer to a previous token (used during code generation).

The output (OUTPUT'STREAM) of the parser is the input (CODE'GEN'INPUT) to the code generator. This module generates the 128 bit microwords (one microword for each source line except Macro, ORG, and Define statements, and comments, for which no code is generated, and NONE statements for which the specified number of no-op lines of code are generated. During this process the code generator checks for semantic errors.

The assembler-driver opens and closes all necessary files, initializes the data structures, reads in the source code, calls the preprocessor, parser, and code generator, causes expanded listings and cross-references to be generated, calculates the values of the parity bits, and arranges the binary output in the form required by the LOLO processor.

Functional Description of the LOLO Preprocessor

Function Preprocessor is the driver for this package. It processes the input line one character at a time, dispatching to a subroutine for each character.

The function first clears TEMP'STRING, the sting in which identifiers are collected.

In the main loop, the function reads the next character off source line INPUT. If this is a multiple blank character, it is converted to a single blank. The character type ACTNUM is then determined by using the numerical value of the character as an index into array ACTION'TABLE. This character type is then used to select the proper subroutine to process the character, which is then called. Upon return of the subroutine, the character is stored into PREV'CHAR for reference during processing of the next character. The actions of the loop are then repeated.

After the source line is exhausted, a check is made to insure that the line ended properly (either with a semi-colon or, in the case of lines containing only a label, with a colon).

If it has been determined (by function Eval'identifier) that this statement is a macro declaration or an ORG or a NONE statement, Preprocessor now calls the proper subroutine to process these statements. The driver then returns to the assembler.

The individual character processing routines are described in the following paragraphs.

Function Space calls ident'or'num to handle identifier processing if the space follows immediately after an identifier or number. No token is generated for the space. Space then returns.

Function Operator calls Ident'or'num if the operator follows immediately after an identifier or number. A token is then generated for the operator, stored into PREP'OUTPUT, and Operator returns.

Function Colon first checks to insure that no tokens have been generated so far for the current line (since the colon is assumed to follow a label which should be the first thing on the line). It then checks to make sure an identifier precedes the colon and calls Ident'or'num to process this identifier. A token for the colon is then inserted into the token buffer, the symbol table entry for the label just processed is initialized (type and value are set) and Colon returns.

Function Semi calls Ident'or'num if the semicolon follows immediately after a number or identifier. A token is then generated for the semicolon and inserted into PREP'OUTPUT. A check is made to insure that no characters follow the semicolon in the source line and then Semi returns.

Function Skip'blanks simply looks for the first non-blank character in line INPUT; it FRETURNS if it doesn't find one else it returns the character.

Function Dot determines whether the dot represents the current line number or whether it indicates field assignment. The function reads the next character from INPUT and if this character is one of '+', '-', '*', '/', ',', ';', or ' ', the dot is assumed to represent the current line number and a constant token for this value is generated and stored in PREP'OUTPUT. Otherwise the dot is assumed to indicate field assignment and function Operator is called. The read pointer for INPUT is backed up one character and Dot returns.

Function Reltn generates tokens for relational operators (=, ≠, >, <, >=, <=). It first calls Ident'or'num to process any number or identifier that immediately preceded the operator. If the character which caused Reltn to be called is '>' or '<', the function checks the next character in INPUT to see if it is '=' (thus making the relation '<=' or '>='), resetting the read pointer if it is not. A token is then generated and stored in PREP'OUTPUT, and Reltn returns.

Function Chrctr, called for alphanumeric characters, simply writes the character on TEMP'STRING and returns.

Function None'init processes NONE statements from the preprocessed text. It first checks to insure that the token following the one for NONE is for a constant. The variables CURRENT'LINE'NUM (which keeps

track of the control store address of the line of code that will be generated for the source line currently being processed) and CODE'LINES (which is the total number of lines of code that will be generated for all source lines processed so far) are adjusted according to this constant.

Function Store'macro stores the text of a macro definition away in a data structure called MACRO'TABLE. It begins by checking the syntax of the macro definition and initializing the symbol table entry for the macro name -- symbol type is set to MAC and symbol value is assigned to be the index in Macro'table of the header for the text associated with this macro. The text, not including the final semicolon, is then stored away. The number of tokens in this macro text is then stored into the macro header, the word preceding the corresponding stored text. The index in MACRO'TABLE indicating the first free word is updated and Store'macro returns.

Function Org'init processes ORG statements from the preprocessed text. It first insures that the token following the one for ORG is for a constant. CURRENT'LINE'NUM is set equal to this constant. Org'init then returns.

Function Ident'or'num simply looks at the first character in TEMP'STRING (where identifiers and numbers were written by Chrctr); after backing up TEMP'STRING's reader pointer one character Ident'or'num

calls the proper subroutine to finish processing the contents of TEMP'STRING -- Eval'number if the first character is a digit else Eval'identifier. After the subroutine returns, the pointers in TEMP'STRING are reset (to zero) and Ident'or'num returns.

Function Eval'number first clears the octal/decimal flag OCT and then calls Cnvrt'str'num to convert the contents of TEMP'STRING into a number. When this subroutine returns, Eval'number creates the pre-processor token for the constant (determining the type according to whether OCT is set or not) and returns.

Function Cnvrt'str'num converts the contents of a specified string into a number which it returns. When conversion begins it is not known whether the number is to be octal or decimal, so both forms are calculated. The function reads one character off the string at a time. If the character is one of '0',...., '7', the following actions are performed: The portion of the octal number previously calculated is multiplied by 8 and the decimal form by 10; the character read from temp'string is converted to a digit by subtracting '0' from it and this digit is added to both the octal and decimal forms. If the digit is 8 or 9, only the decimal form is computed and flag DEC is set (to indicate that this must be a decimal number.

If the character read from temp'string is a 'B' (indicating that the number is octal) the following is done: a check is made to insure flag DEC is off; if not, an error message is generated. The octal form

calculated is assigned to variable NUM, flag OCT is set (for use by Eval'number) and a check is made to see if a digit follows the 'B' in TEMP'STRING. If so, NUM is modified by multiplying it by 8 the number of times specified by this digit. Checks are then made to insure that TEMP'STRING is now empty and that the calculated number will fit in a 16 bit register. NUM is then returned.

If the character read from temp'string is a 'D', actions similar to the above are taken except that the decimal form is returned.

If the character from temp'string is none of '0',...., '9', 'B', or 'D', an error message is output.

Function Name'lookup performs the same function as the utlity call Abr'lkp (abbreviated lookup). It searches for a specified string (first argument) among the elements of a string array (second argument) of a given dimension (third argument). If successful, it returns the index of the string in the array. Otherwise it FRETURNS. It works in this manner:

It assigns each string in the string array, in turn, to S1. If the length of S1 and TS (the string being searched for) differ, no further comparison is performed between these two strings. If the lengths are the same, the strings are compared character by character. As soon as two corresponding characters fail to match, the comparison is aborted and the next element of the array is tested. If the strings match, the index is returned.

Function Eval'identifier processes identifiers. First it checks to see if the identifier is a keyword by calling Name'lookup with the identifier being evaluated (in TEMP'STRING) and KEYWORD'ARRAY. If it is found that this identifier is a keyword, a check is made to see if it is the word MACRO, REGISTER, SFUNCTION, PARAMETER, BCONDITION, ORG, or NONE. If this is the case and the word does not occur within a macro definition, one of several flags is set to indicate special processing is required later on. The token is then created for the keyword and the function returns.

If the identifier is not a keyword, a check is made to see if it is a register name and if so a token is created and the function returns. If it is not a register name, then it is determined whether it is a field name; if so, a token is created and Eval'identifier returns.

If the identifier is neither a keyword, register name, nor field name, it is assumed to be a user defined word and a check is made to see whether there is already a symbol table entry for this word. This check is made by calculating a hash code for the identifier and indexing into the symbol table through the hash table BUCKETS. The length of the identifier being searched for is compared with each identifier in the symbol table which has the calculated hash code, in turn. If the lengths are not equal, the next word is tried. If the lengths are equal, a word by word comparison is performed. As soon as corresponding words fail to match, comparison with this particular word is aborted.

If the identifier is found to be in the symbol table a check is made on the type of the symbol. If it is a macro name, the first two tokens

in this macro's definition are checked to see if this macro expands into DEFINE statement. If so, the proper flag is set to indicate special processing is required later on. No matter what the symbol type was, a token is created and the function returns.

If the identifier being evaluated is not already in the symbol table, an entry is made for it. This consists of entering the name in table IDENT'STORAGE and setting the type of the corresponding symbol'table entry to IDENTIFIER. A token is created and the function returns.