Rochester Institute of Technology
RIT Digital Institutional Repository

Theses

1980

The CMS-2 subset compiler

J. T. Badura

Follow this and additional works at: https://repository.rit.edu/theses

Recommended Citation
Badura, J. T., "The CMS-2 subset compiler" (1980). Thesis. Rochester Institute of Technology. Accessed from

This Thesis is brought to you for free and open access by the RIT Libraries. For more information, please contact repository@rit.edu.

Master of Science in Computer Science

Thesis Approval Form

This is to certify that _____ has submitted a thesis entitled:

The CMS-2 Subset Compiler

to the faculty of the School of Computer Science and Technology in partial fulfillment of the requirements for the degree of Master of Science.

Approval: _____ _____
(thesis advisor) (date)

_____ _____
(committee member) (date)

_____ _____
(committee member) (date)

THE CMS-2 SUBSET COMPILER

Submitted by:

J. T. Badura

G-926956

TABLE OF CONTENTS

THE CMS-2 SUBSET COMPILER

INTRODUCTION

The CMS-2 compiler represents the culmination of an effort to construct a production compiler for military applications. The material that follows is an effort to render succinct explanations of the structure and rationale behind the compiler design.

The thesis contained herein is submitted in fulfillment of the Master of Science degree in Computer Science at Rochester Institute of Technology.

HOST SYSTEM

The CMS-2 compiler was developed on the the DEC-10 at MCDONNELL DOUGLAS in Huntington Beach, California. The DEC-10 runs under a TOPS-10 Operating System using a KL-10 processor. The machine possesses 256K words of memory.

A variety of software tools were used for the construction of the compiler. For example, the front end of the compiler was written using the Metatranslator, a compiler-compiler that generates FORTRAN. A few of the early semantic routines were written in FORTRAN, while the remainder of the compiler was written in FRAN, which is a structured pre-processor for FORTRAN. Lastly, a string manipulation routine was written in MARCO-10, the assembly language of the DEC-10.

THE CMS-2 LANGUAGE

CMS-2 is a Navy language that is specialized for real time applications. One notable feature of this language is its capability for fixed point operations.

The only recognized CMS-2 compiler is the one developed by the United States Navy. The Navy's compiler will compile for either the AN/UYK-7 or AN/UYK-20(V) computers. The compiler described in this report compiles for a subset of the original CMS-2 language.

TARGET MACHINE

The target machine for the compiler is the MDAC 476CX microprocessor, which is essentially a single accumulator-one address machine. The 476CX possesses 8 bit data words and 16 bit addresses. The accumulator is register A , which may be used for single precision operations. For double precision, the register pair AB is used. The machine contains A' and B', which are shadow registers of A and B respectively. In other words, when A is loaded, register A' receives the previous result of A

before the load. The same relationship holds for B and B'. As for hardware, the microprocessor possesses a 181 ALU chip. Moreover, the 476CX is classified as an isolated I/O machine.

THE CMS-2 SUBSET COMPILER

PURPOSE OF THE CMS-2 SUBSET COMPILER

The CMS-2 compiler is essentially a cross compiler that produces an input file to the 476CX assembler, which is also hosted on the DEC-10. The assembler generates an object file that inputs to the 476CX linkage editor, producing a relocatable load module. This load module is then used to program PROM chips for the 476CX microprocessor. In essence, the intended use of the compiler is to act as a partial translation step in multiple cross translations.

COMPILER DESIGN OBJECTIVES

The compiler was designed according to six fundamental design goals, which are listed below:

1. The compiler will generate optimal assembly code that will execute as fast as possible. This was the prime objective, since the target machine will be used for real time applications. Therefore, the compiler will generate the minimum number of assembly language instructions and take advantage of those instructions that are less expensive.

2. The compiler will optimize storage allocation of variables and temporaries. The minimum number of temporaries will be generated while the allocation positions of variables will be optimal for code generation.

3. The compiler will be flexible. In the future, if it is desired to generate assembly language for a new target machine, only the third and fourth pass will be changed. (In all probability a fourth pass will not be needed since it performs code generation that is unique to the 476). If a language other than CMS-2 is needed to be translated into 476CX assembly code, only the first pass will be rewritten. Therefore, the CMS-2 compiler will be reconfigurable for future needs.

4. The compiler will generate assembly code that is clear and readable to the user. User names for variables and labels will be carried through to the assembly language file. Compiler generated labels and temporaries will be distinguishable from user labels and variables. The compiler will also aid the user by generating some comments.

5. Compilation errors will be easily debugged. The compiler will produce understandable error messages rather than confusing error codes. Warning messages will also be output to aid the programmer.

6. Compiler maintenance will be easily performed. If an internal compiler error occurs, the internal compiler error number will be printed out, indicating the cause and location of the error. The compiler will also have the capability of parsing debug flags set by the systems programmer for the purpose of displaying tables, arrays, and other debugging aids.

Overall, the optimizations of the CMS-2 compiler include those that are user independent. It will be assumed that the user is an experienced programmer who possesses an understanding of CMS-2 as well as the 476CX assembly language. Optimizations will center on improving inefficient intermediate language quadruples (quads) produced by the parser and machine dependent optimizations intrinsic to the assembly language. The objective of this project is not to optimize source code that the user could improve by rearranging the CMS-2 statements.

OVERVIEW OF COMPILER STRUCTURE

The diagram in Figure 1.1 illustrates the flow of source code, intermediate code, and target language through the system:

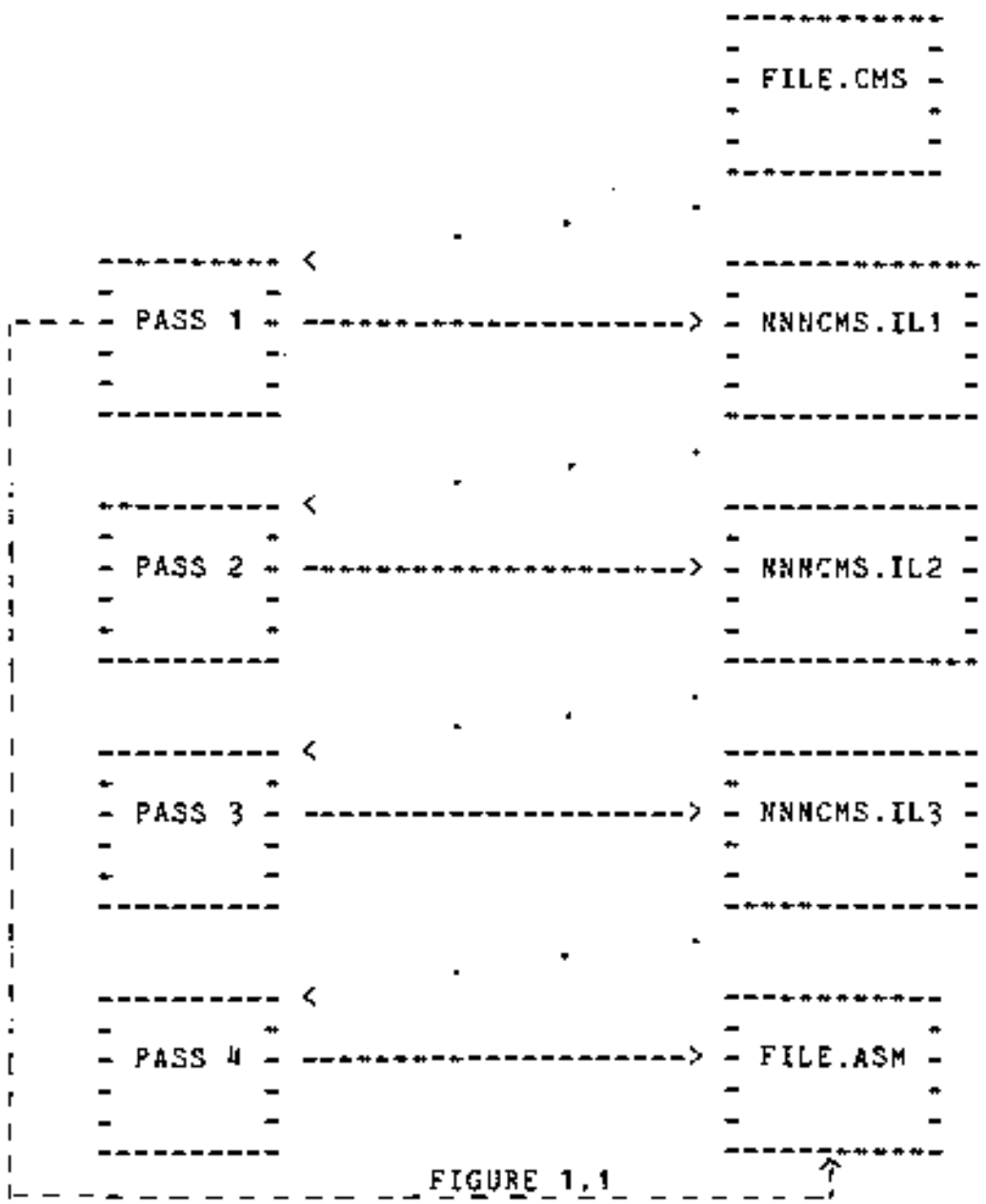


FIGURE 1.1

The CMS-2 source statements are located in FILE.CMS. This file is read in by pass 1, which performs lexical and syntactic analysis. The output of this pass is an intermediate language in the form of quadruples (quads). Pass one also generates

assembly language statements that reserve storage for variables of the SYS-PROC. Pass 2 reads in the quads and performs machine independent optimizations on them. The output of the second pass is a set of optimized quads. Pass 3 reads in the optimized quads and generates assembly code for all the quads except the following:

1. Branches
2. Subroutine calls
3. Entry to procedures or functions
4. Return from procedures or functions

The quads listed above are carried through to pass 4, which generates assembly language for them. Pass 4 reads in the assembly language generated by pass 3 and merely writes it out to FILE.ASM.

The temporary files generated by pass one, two, and three have the extensions IL1, IL2, AND IL3 respectively. The file name for all temporary files is NNNCMS. The NNN receives the job number that is assigned to the user upon logging onto the account.

The compiler utilizes two files that contain global variables in common. CMSP.COM contains the common variables for all of the user written routines plus some common parser variables. CMS2.COM contains the common variables for the parser; this file is automatically generated by the Metatranslator.

The compiler generates code for each SYS-PROC using the following calling sequence. Initially, pass 1 is called. When the end of the SYS-PROC is reached, pass 2 is called. When an end of file condition is reached, pass 3 is called. Upon end of file, pass 3 calls pass 4. When pass 4 finishes processing, it returns to pass 3, pass 2, and pass 1. At this point a cross reference map is generated by the first pass. Next, pass 1 clears out the local symbols of the symbol table and loads in the new symbols. By purging the symbol table after each SYS-PROC is processed, the scope of the variables is maintained.

DEBUGGING FEATURES

The debugging features of the compiler are intended for maintenance systems programmers rather than general compiler users. Debugging centers around the usage of debug flags, which optionally coexit with CMS-2 statements in the source file. These flags are parsed like regular CMS-2 statements but do not generate quads that produce assembly code. Instead, they set flags which perform various functions.

Debug flags can be turned on and off at will. In fact, more than one debug flag can be turned on or off at one time. For example, debug flags 5 and 7 are turned on when an input card beginning with '.$+57' is encountered.

All of the debug flags of the Metatranslator are available as well as a few others. The most useful include:

.$+57	STEP UP AND STEP DOWN TO TRACE PARSE
.$+9	ANNOUNCE PRIMITIVE TERMINAL ELEMENTS
.$+K	DUMP THE PARSE IMAGE BEFORE PARSING
.$+L	PAUSE (DOES NOT NEED TO BE TURNED OFF)
.$+P	PRINT THE INTERMEDIATE LANGUAGE AFTER PASS 2 OPTIMIZATIONS HAVE BEEN PERFORMED
.$+S	COMMENT LINES CONTAINING DASHES ARE INSERTED IN THE ASSEMBLY CODE TO SEPARATE THE CODE GENERATED FOR EACH QUAD
.$+T	TRACE AND EXIT TO/FROM FORTRAN ROUTINES
.$+V	PRINT OUT PASS 2 OPTIMIZATION STATISTICS
.$+W	PRINT OUT GOTO TO GOTO TABLE
.$+X	PRINT OUT SOURCE AND IL QUADS BEFORE PASS 2 OPTIMIZATIONS HAVE BEEN PERFORMED
.$+Y	PRINT OUT LABEL AND BRANCH TABLES
.$+Z	PRINT OUT CONTENTS OF REGA DESCRIPTOR IN PASS 3

One of the advantages of using debug flags is that it is not necessary to recompile the routines composing the CMS-2 compiler in order to display debug information.

OVERVIEW OF THE METATRANSLATOR

The Metatranslator is a user-oriented syntax-directed translator generator that is machine independent over a large class of machines.

The Metalanguage is the language that the Metatranslator uses to define a programming language. The Metalanguage is very similar to BNF. A production in a grammar corresponds to a Metalanguage definition statement, while a nonterminal corresponds to a metalinguistic variable, which is referred to as an mlv.

The compiler may be considered a translator system that is composed of the following elements:

1. Control Driver This is the main program of the compiler, which is written in FORTRAN by the user. This program initiates translation.

2. Q-Routines These are external support routines that are used by the Metatranslator or user routines. Their functions include input, output, text editing, packing, unpacking, conversions, and branching from mlv to mlv.

3. Semantic Routines These are user written routines that perform the semantic actions of the parse. These routines determine the output of each metalinguistic variable.

4. Parser This is a recursive descent parser generated by the Metatranslator. The parser is defined using the Metalanguage.

5. Procedure Executor This is a routine generated by the Metatranslator that executes deferred procedures. A deferred procedure may be though of as a semantic action routine that can be user defined or built-in (as in the case of TEXT).

Metalinguistic definitions are used to define the parser. Each metalinguistic variable begins with a dollar sign. The format of a metalinguistic definition is the following:

```
$mlv-name .= metalinguistic definition.
```

The first mlv in a language definition defines the highest syntactic level, referred to as the head node. Each mlv corresponds to a node in the implicit parse tree. For example:

```
$UNCONDGOTO .= $GOTO,$INTEGER.
```

defines a FORTRAN unconditional goto.

The Metatranslator generates a block of FORTRAN code with a

label to denote an entry point to a nonterminal (mlv). The last several digits of the label denote the identification number of the mlv. When an mlv steps down to another mlv, this is performed by a goto rather than a subroutine call. Because recursion is not supported in FORTRAN, the generation of subroutines for each nonterminal is not feasible. Therefore, the Metatranslator maintains its own stack of information about the parse and return points. The Metatranslator uses a stack called Qctab to record the state of the parser at each point. Some of the major variables include the following:

Cursor. This variable points to the current character in the image buffer.

Qmlv. Denotes the mlv id number of the node which is being entered (stepped down into).

Qsave. Denotes the label number of the mlv which the parser will return after the parse of the current mlv is completed.

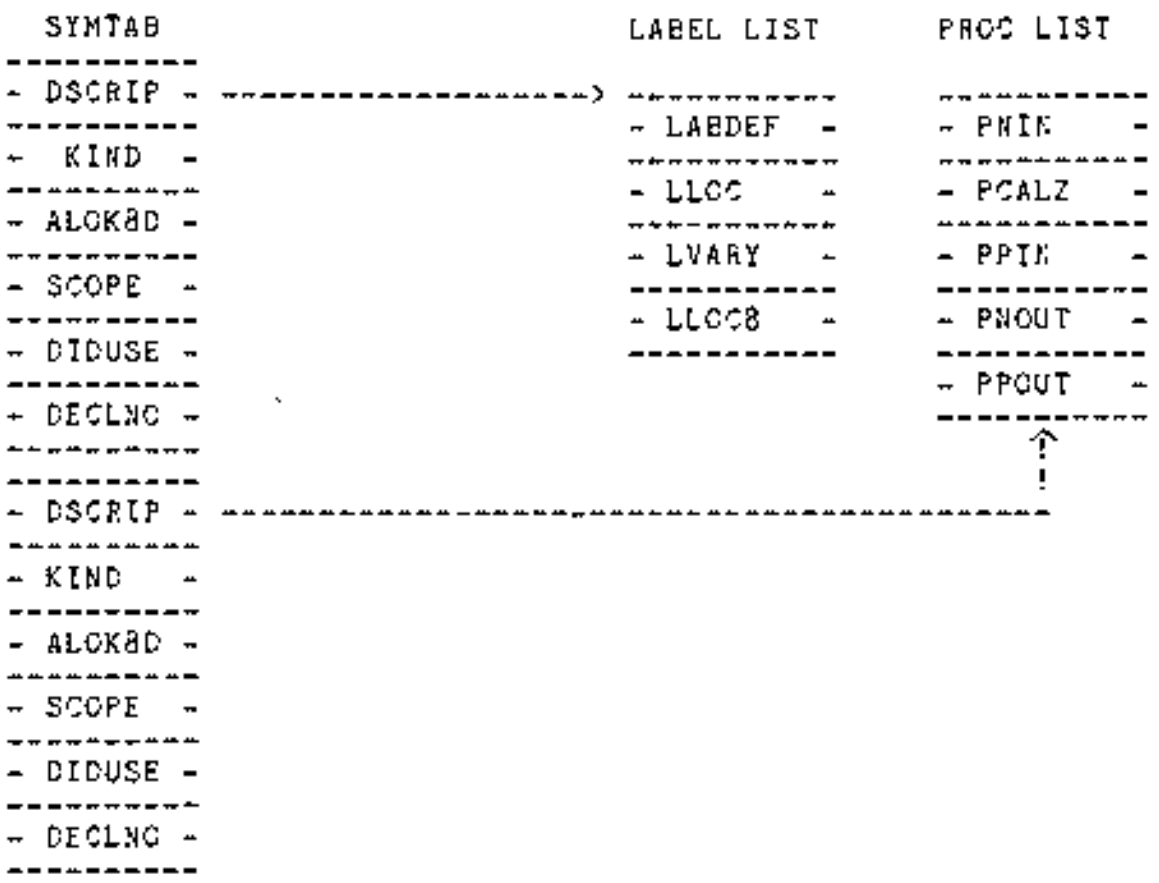
The Metatranslator generates a parser with the same name as the head node. At the beginning of translation, the driver calls the head node. Each node steps down to the next successive node as defined by the metalinguistic definitions. When the parse finally steps back up to the head node, translation is completed and the system returns to the driver.

Each time the Metatranslator steps down to the next node, the status of the parse is pushed onto Qctab. When it steps back up to the previous node, these attributes are popped from the stack. If the parse succeeds, a variable called truth is set to one, else it is set to -1.

THE CMS-2 SUBSET COMPILER

SYMBOL TABLE

The symbol table is basically a hash table which points to a node within a descriptor list. This node contains additional information about the hashed symbol. A section of the symbol table is shown below:



SYMTAB is the name given to the hash table representing the symbol table. Each symbol will hash into a bucket in SYMTAB. If a collision occurs, a pointer associated with each bucket, Ilink, is set to point to a free hash record. The Metatranslator C-routines perform the storage management. If no collision occurs, Ilink is set to zero.

Each hash bucket contains six attibutes that are common to all of the symbols; they are as follows:

DSCRIP	This is a pointer that points to a node in a descriptor list
ALLOK8D	This is a flag that indicates whether the symbol has been allocated. It is set to 1 if the symbol has already been allocated, otherwise 0
SCOPE	This explains the scope for the variable. The values are as follows: 1 External symbol 2 Local symbol 3 Global symbol 4 Entry point
DIDUSE	This is a flag that is 1 if the symbol has been used in a SYS-PROC, otherwise 0.
DECLNO	This attribute is used for the cross reference table. It is the line number on which the symbol is declared.
KIND	This attribute indicates the symbol type (variable, label, procedure, etc.)

After the symbol is hashed into one of the buckets, a node is allocated in the descriptor list. DESCRIP is then set to point to that node in the list. For example, a variable points to a node in the variable list and a procedure name points to a node in the procedure list. Although the node also points to the next node and the preceeding node, these pointers are not used. Whenever a node is allocated, the storage management Q-routines automatically allocate it as a node in a doubly linked list. However, for the symbol table, these forward and backward pointers are not necessary. The local symbols in the symbol table are released by returning all of the nodes in the linked lists back to the storage pool.

The following lists are used to contain descriptor information about hashed symbols:

LISTNAME	FUNCTION
VRBL	Variable list
LABEL	Label list
PROC	Procedure list
FUNC	Function list
TABLE	Table list
FIELD	Field list
EQUALS	Equals tag list

For example, the procedure list contains the following descriptors:

PNIN	Number of input parameters
PCALZ	Flag to indicate whether this procedure calls another procedure
PPIN	Pointer to input parameters
PNOUT	Number of output parameters
PPOUT	Pointer to output parameters

All of the data structures used in pass 1 are generated and manipulated by the Metalanguage. Therefore, the stacks, hash tables, and linked lists in the first pass have been defined using built-in Metalanguage declaratives. There are also Metalanguage statements that will add and delete items from linked lists, add items to hash tables, and test whether an item is already in a table.

Other data structures that pass 1 utilizes include stacks that are used during parsing and a hash table called RESERVED that contains all of the reserved words of the CMS-2 language.

PASS 1

LEXICAL ANALYSIS

One of the functions of a lexical analyzer is to read the source program characters into a buffer. The Metatranslator performs this action through various Q-routines. Qinput reads in the characters and places them into a buffer. As the characters are read in, Qconv converts the characters and integers into the internal character-integer code of the Metatranslator. This conversion helps to preserve the machine independence of the first pass.

The second role of a lexical analyzer is to partition an input string into a stream of logical units called tokens. Tokens generally include items such as keywords, identifiers, constants, and punctuation symbols.

The Metatranslator incorporates lexical analysis into the syntactic analyzer. One can easily use BNF to describe the syntax of a symbol. For example, the following defines the syntax of an identifier.

```
<identifier> ::= <letter> / <identifier><letter> /
                 <identifier><digit>

<letter> ::= a / b / c /...z
<digit>  ::= 0 / 1 / 2 / 3 / 4 / 5 / 6 / 7 / 8 / 9
```

A possible Metalanguage definition for an indentifier is as follows:

```
$IDENTIFIER .= 1 TO 8 LETMERICS.
```

The Metatranslator has reserved keywords that represent various subsets of the entire character set. Moreover, these keywords are used to define tokens. The keywords represent the the terminals of a metalinguistic definition; they are shown below.

KEYWORD	TYPE	CHARACTER TYPE
LETTER	1	A-Z
DIGIT	2	0-9
ALMERIC	3	A-Z,0-9
LETMERIC	4	Letter, followed by digits or letters
SPACE	5	space or blank only
SPECIAL	6	all FORTRAN special characters
NONQUOTE	7	all except quote

OPERATOR	8	arithmetic operators (+,-,*,/, AND **)
CHARACTER	9	all
NONSPACE	10	all, except space or blank
HEX	11	0-9, A-F
BINARY	12	0 OR 1
OCTAL	13	0-7

The Q-routine Qterm has the responsibilty of identifying terminals. Suppose the following metalinguistic definition is being parsed:

```
$INTEGER .= 1 TO 5 DIGITS.
```

In this case, Qterm compares each character under the cursor with a digit. The parse succeeds and thus terminates when the maximum number of digits has been recognized. The parse fails if the minimum number of digits is not encountered.

A syntactic definition may also include keywords. For example, the metalinguistic definition of a goto statement may be the following:

```
$GOTOSTMT .= 'GOTO',$LABEL.
```

When the Metatranslator generates the parser, the keyword enclosed in quotes is placed into a literal table called Qlvect. When the parser is parsing a card image and the present mlv is a $GOTOSTMT, then it is expected to recognize a 'GOTO' keyword. Therefore, each character in the card image is compared with the 'GOTO' in the literal table. If they are equal, the next mlv, $LABEL, is parsed.

SYNTACTIC ANALYSIS

The intricate mechanism of parsing is best described by an example. The following metalinguistic definitions are similar to those in the CMS-2 parser.

```
$SIMPLESTMT .= $GOTOPHRASE,$EOL.

$GOTOPHRASE .=
        'GOTO',
        $NAME,
        BEGIN
            IF $NAME IN SYMTAB,
            BEGIN
                IF KIND EQ LABELK,
                 .L IL(ILJ,KEYLAB,SYMPTR,EJECT)
            //   .L IL(ILERR,12,ERR,$NAME,EJECT)
            END
        //      ETC...
        END.

$EOL .=
        '$',
        CALL CMS2EX,
        CALL MORE.
```

The syntactic definition states that a simple statement is composed of a goto phrase followed by an end of line marker. A goto phrase is composed of the literal 'GOTO' followed by a name. If the name is in the symbol table and kind equals label kind, then a link is established to the user-defined procedure called IL. The first reference to IL links IL with the arguments of an unconditional jump quad. Otherwise, link to IL containing the arguments of an error quad. When a link to a deferred procedure is established, the procedure is not executed. Instead, its arguments are stacked in Qctab. When the parsing of $GOTOPHRASE is finished, $EOL is then parsed. This mlv parses a dollar sign and then calls CMS2EX, which is the name of the procedure executor generated by the Metatranslator. The procedure executor executes the deferred procedure IL, which unstacks the quad arguments from Qctab and writes them out to the intermediate language file. Next, MORE is called which reads in the next card image.

Boolean conditionals in CMS-2 are parsed in the most efficient manner possible. Let Boolean factors be defined as items separated by OR's and Boolean term be items separated by AND's. If any of the Boolean factors are true, then a branch to the true exit is made without evaluating the other Boolean conditonals. For example, suppose the following Boolean expression is encountered:

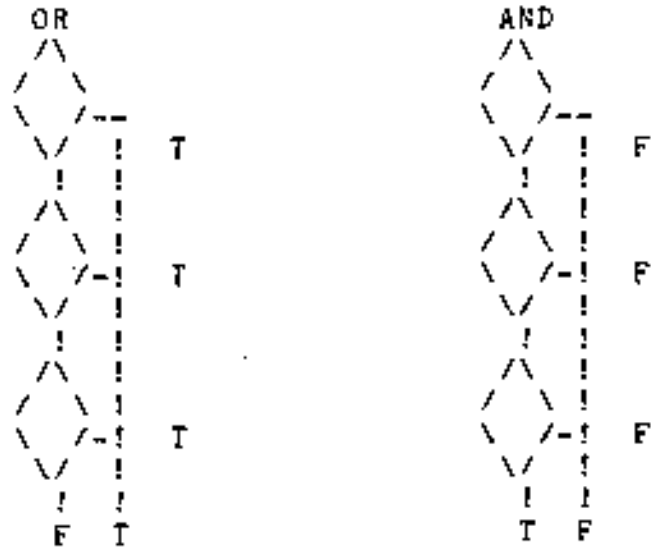
```
IF V1 EQ 5 OR V2 EQ 6 THEN SET V3 TO V3+1 $
```

The following quads are generated. The comments do not appear in the actual listing but have been added here for clarity.

```
        NE,V1,I5,G4      ;IF FALSE, FALL THROUGH TO NEXT CONDITONAL
        J,G1             ;OTHERWISE JUMP TO TRUE EXIT
G4:
        NE,V2,I6,G5      ;IF FALSE, FALL THROUGH TO NEXT CONDITIONAL
        J,G1             ;OTHERWISE JUMP TO TRUE EXIT
G5:
        J,G2             ;JUMP TO END OF STATEMENT
G1:                      ;TRUE EXIT
        +,V3,I1,T1
        SET,T1,V3
G2:                      ;END OF STATEMENT
```

If the conditional statement is an if-then-else construct, then the jump to the end of the statement is essentially a jump to the else portion.

If any Boolean terms are false, then a jump to the false exit is performed immediately, otherwise the next Boolean term is tested. The diagram below illustrates the short circuiting of boolean conditionals. The diamonds represent each Boolean conditional test while the straight line downward represents the shortest path.



The problem with this method is that it generates goto's to goto's. For Boolean factors the algorithm generates conditonal quads to jump to the next conditional test and an unconditional jump quad to exit early (for Boolean terms it is the other way around). The last conditional quad jumps to a label which then jumps to the end of the statement; this is the goto to goto. The goto to goto code is optimized during pass 2.

Arrays in CMS-2 have an index origin of zero and are stored in

column major form. The basic equation used for array referencing given an array A(I,J,FLD) is as follows:

base+((0*c+j)*r+i)*width+offset where

base = address of the array
c = number of columns
r = number of rows
width = the total number of words for all of the fields
offset = the number of words up to the field being referenced

Suppose FOX is an array that is declared with the dimensions 3 by 4 by 2. The algorithm used to generate code to reference FOX(1,2,0,FOX2) is as follows:

1. Push on a stack a pointer to each subscript. The top of the stack now points to the field FOX2 while the bottom points to the 1.

2. Pop the stack so that the pointer to FOX2 is in a variable and the top of the stack points to 0.

3. Generate a SET quad that places a 0 into a temporary called T1.

4. Generate a quad that multiplies T1 by the number of elements of the dimension pointed to by the top of the stack, storing the result in T1. Add the subscript pointed to by the top of the stack and store the result in T1.

5. Pop the stack. Repeat step 4 until the stack is empty.

6. Generate a quad that multiplies the width times T1. The result is stored in T1. Add the offset to T1 and store the result in T1.

The generation of quads for accessing arrays stored in column order form requires that the subscripts enter in the calculations in reverse order. Because a parse scans from left to right, it was necessary to stack the subcripts as described.

Parameter transmission to subroutines and functions is basically a call by value with copy restore. Before the call, quads are generated that copy the arguments into the input parameters and the call quad is generated. If the item being called is a procedure, then the quads are generated to copy the output parameters into the output arguments. For example, the calling sequence for a procedure called PROC(X,Y,Z) with A, B, C as arguments is illustrated by the following set of pseudo quads:

```
        SET A,X          ;COPY IN FIRST INPUT PARAMETER
        SET B,Y          ;COPY IN SECOND INPUT PARAMETER
        CALL,PROC
        SET,Z,C          ;COPY BACK OUTPUT PARAMETER
```

If PROC was a function called PROCF, then the calling sequence is the following:

```
        SET A,X          ;COPY IN FIRST INPUT PARAMETER
        SET B,Y          ;COPY IN SECOND INPUT PARAMETER
        CALL PROCF,T1
```

If the function returns a single precision result, then the result is automatically loaded into register A. If it is a double precision function, then register pair AB is used. The temp T1 indicates a temporary into which the result of the function may be stored if there is no next use.

It is also possible to pass arrays to a function or procedure; this is achieved using call by address. The function or procedure is defined using CORAD(ITBL), where ITBL is some indirect table. Indirect tables may be thought of as templates which reserve no storage (except the double word containing their address). To pass an array, a call is performed using CORAD(TBL), where TBL is a direct table that actually exits. The CORAD intrinsic function passes the address of the direct table.

STORAGE ALLOCATION

The allocation of variables is termed pass 1.5 because it is performed after parsing (pass 1) and before quadruple optimization (pass 2). Moreover, during this pass assembly language necessary for allocation of variables is written to the ASM file. The code for this pass is written in the Metalanguage. A mlv called $ALLOCATEGLOBAL allocates all of the global variables, which are contained in a SYS-DD. For example, suppose the variables XYZ and ABC were found in a SYS-DD called GLOBAL. Pass 1.5 would generate the following assembly code:

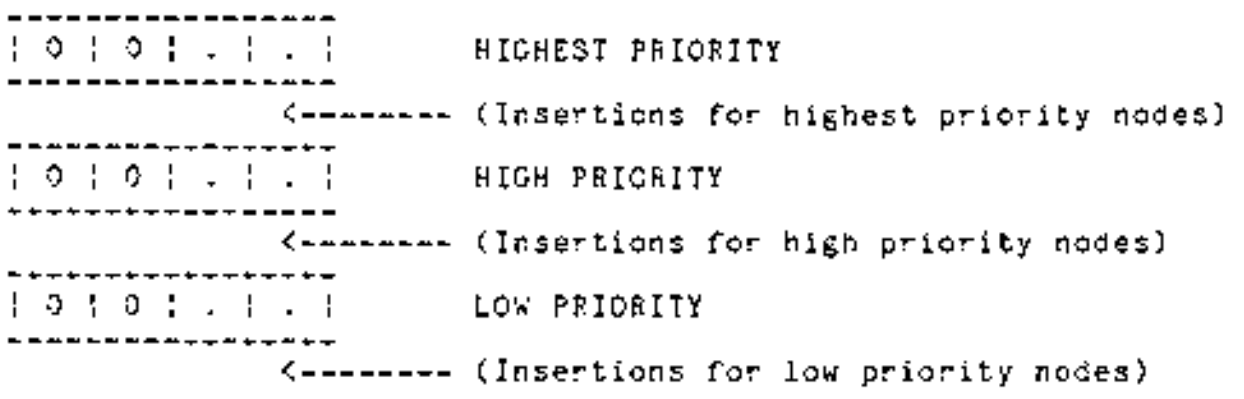
```
        MODULE  GLOBAL
        DSECT
ENTRY   ABC
ABC     RES     1
ENTRY   XYZ
XYZ     RES     1
        END
```

All global variables are allocated in a module with the same name as the system data design. The ENTRY assembler directive indicates that the variable is a global entry point and its

definition follows. If ABC is used in the procedure, then it is declared with the EXTERNAL directive indicating that it is an external variable that has been declared elsewhere.

The mlv called $ALLOCATELOCAL performs the allocation of local variables. If a CMS-2 statement is written in which two local variables are used in a binary operation, the most efficient assembly code is generated when the two variables are allocated next to each other in storage. The section on pass 3 explains why more optimal code is generated in this case.

Pass 1 keeps a doubly linked list called INTEROP which contains pairs of local variables that would benefit from adjacency. The list starts out as three nodes, which serve as markers for priority levels.



The allocation scheme is based on the premise that the variables that are allocated at the beginning of the list will have a greater chance of being next to each other. Therefore, those variables with a high priority will be allocated at the head of the list and those with low priority will be allocated at the rear of the list. The highest priority is assigned to those variable pairs that are in a vary block that is nested inside another vary block. These variables are apt to be used the most often. High priority is assigned to variables within a vary block, while low priority is given to variables not within a vary block. When a new set of binary operators is encountered, the variable components are inserted into the list according to their priority level as shown.

Suppose variables X1 throuth X8 are local. INTEROP is initially set to (0,0);(0,0);(0,0) ,which are the three priority levels. If X1+X7 is found, this will result in INTEROP being (0,0);(0,0);(0,0);(X1,X7). If X1+X8 is encountered inside a VARY block, it is added under the high priority level; this results with INTEROP = (0,0);(0,0);(X1,X8);(0,0);(X1,X7). Suppose after several more trials, INTEROP is equal to (ignoring the zeroes) (X1,X8);(X1,X3);(X1,X7);(X5,X7).

Pass 1.5 builds an allocation model in which sequences requiring

adjacent allocation are separated by zeroes. MODEL is a doubly linked list used for the final sequencing of the variables. Initially, MODEL = 0,0. Considering the first pair in INTEROP, MODEL = 0,X1,X8,0. The second pair causes MODEL to be 0,X3,X1,X8,0. The third pair cannot fit into the model and is therefore ignored. The fourth yields MODEL = 0,X3,X1,X8,0,X5,X7,0. When INTEROP is exhaused, X2,X4, and X6 are allocated linearly, thus giving the order:

```
X3      RES     1
X1      RES     1
X8      RES     1
X5      RES     1
X7      RES     1
X2      RES     1
X4      RES     1
X6      RES     1
```

Lastly, pass 1.5 writes the assembly code it has generated directly to the assembly language file.

CROSS REFERENCE TABLE

A cross reference table is generated by pass 1. The cross reference table lists global elements of a system data design and the local elements of a SYS-PROC. A mlv called $CROSSREF generates the cross reference information. The format of the listing is as follows:

NAME KIND TYPE LINE ADDR

Under the name heading lies the name of the element. Kind specifies the kind of data element; the following possiblilites include PROC (procedure), EQU (equals declaration), VRBL (variable), TABLE, FIELD, OR FUNC (function). The type attribute indicates the type of variable listed. Type can be I (integer), R (real), A (arithmetic fixed point), or B (Boolean). Line indicates the line on which the element was declared. Addr indicates the address of the element. If the element is a variable, then it represents the hex bias from the DSECT. If it is a name enclosed in parenthesis, then it refers to the name of the table that the item (a field) belongs. If it is '**', then it is an indirect table. Another column preceeds the name column; it lists the scope qualifier for each element. The keys include S (global scope), N (entry), X (external), and blank indicates local scope.

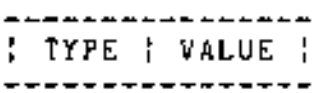
INTERMEDIATE LANGUAGE

The following is the intermediate language generated by pass 1:

QUAD	MEANING
L.ptr	User label
J,OP1	Jump to OP1
JF,OP1	Jump on false to OP1
JT,OP1	Jump on true to OP1
SRC	CMS-2 source statement
COM	Comment
DIR	Direct code flag
+,OP1,OP2,OP3	OP3 <- OP1+OP2
-,OP1,OP2,OP3	OP3 <- OP1-OP2
*,OP1,OP2,OP3	OP3 <- OP1*OP2
/,OP1,OP2,OP3	OP3 <- OP1/OP2
OR,OP1,OP2,OP3	OP3 <- OP1.OR.OP2
AND,OP1,OP2,OP3	OP3 <- OP1.AND.OP2
XOR,OP1,OP2,OP3	OP3 <- OP1.XOR.OP2
EQV,OP1,OP2,OP3	OP3 <- OP1.EQV.OP2
SET,OP1,OP2	OP2 <- OP1
SHR,OP1,OP2,OP3,OP4	Perform a type OP1 shift upon OP2 for OP3 number of times. The destination is OP4.
ABS,OP1,OP2	OP2 <- ABS(OP1)
CALL,OP1,OP2	Call procedure or function OP1. If a function, result is left in OP2.
ENTER,OP1	Entry to function or procedure
RETF,OP1	Return from function; the returned value is left in OP1.
RETP	Return from procedure
CORAD	Call to CORAD
ERR,OP1,OP2	Print out error message number OP1 with severity level OP2.
EOF	end of file flag
~,OP1,OP2	OP2 <- ~(OP1)
COMP,OP1,OP2	OP2 <- COMPLEMENT(OP1) (1's complement)
EQ,OP1,OP2,OP3	IF OP1 = OP2 GOTO OP3
NE,OP1,OP2,OP3	IF OP1 != OP2 GOTO OP3
GT,OP1,OP2,OP3	IF OP1 > OP2 GOTO OP3
GE,OP1,OP2,OP3	IF OP1 >= OP2 GOTO OP3
LT,OP1,OP2,OP3	IF OP1 < OP2 GOTO OP3
LE,OP1,OP2,OP3	IF OP1 <= OP2 GOTO OP3
C.no.	Compiler generated label

NEVER	Inaccessible code
AGET,OP1,OP2,OP3	Read from an array OP3 <- OP2[OP1] (OP2 IS THE SUBSCRIPT) (Note: there is no base register in the 476. The notation is used in the abstract sense.)
APUT,OP1,OP2,OP3	Write to an array OP3[OP2] <- OP1 (OP3 IS THE SUBSCRIPT)
IGET,OP1,OP2,OP3	Read from an indirect array
IPUT,OP1,OP2,OP3	Write to an indirect array
TOAD,OP1,OP2,OP3	Set the address of an indirect array.
DEBUG,OP1	Set debug flag
KILL	Generate no code in pass 3

The following is the format for the operands in the intermediate language:



Each operand is composed of two elements: type and value. The value element is either an integer or real number. The type element determines the meaning of the value element. The eight different type elements are listed below.

SYMBOL	ENUMERATION VALUE	MEANING OF VALUE
I	1	INTEGER
R	2	REAL NUMBER
V	3	POINTER TO VARIABLE
T	4	TEMPORARY NUMBER
L	5	POINTER TO USER LABEL
F	6	POINTER TO FUNCTION
P	7	POINTER TO PROCEDURE
C	8	COMPILER LABEL NUMBER

PASS 2

Pass 2 performs two basic functions:

1. Construction of a label reference array
2. Machine independent optimizations upon quadruples

The label reference array is a one dimensional Boolean matrix which is initialized to all zeroes at the beginning of pass 2. Each position corresponds to a compiler generated label number. When a conditional or unconditional branch quad is encountered, the array is updated with a one in the location corresponding to the destination of the branch. The purpose of the label reference array is to inform the third pass which compiler labels have been actually referenced. This table is necessary because optimization during pass 2 may leave a few compiler labels that are no longer needed.

Two types of machine independent optimizations are performed in pass 2. The first is termed pinhole optimization, which is defined as the transformation or mapping of a quad Q into an optimized quad Q' by some optimizing function F.

$$Q \xrightarrow{F} Q'$$

The second is termed peephole optimization, in which two or more quads are optimized at a time.

PINHOLE OPTIMIZATIONS

The three optimizing functions that are used for pinhole optimizations include the following:

1. Logical Simplification
2. Algebraic Simplification
3. Optimal Commuting

Table 2.1 illustrates the list of pinhole optimizations used.

TABLE 2.1

V = ANY VARIABLE
T = ANY TEMPORARY
C = ANY CONSTANT(REAL OR INTEGER)
VAR = VARIABLE OR TEMPORARY
CON = COMPILE TIME COMPUTATION
!= MEANS NOT EQUAL TO

INITIAL QUAD	OPTIMIZED QUAD	
Logical Simplification		
GT,C,C,LABEL	DELETE	IF C NOT > C
GT,C,C,LABEL	J,LABEL	IF C > C
LT,C,C,LABEL	DELETE	IF C NOT < C
LT,C,C,LABEL	J,LABEL	IF C < C
LE,C,C,LABEL	DELETE	IF C NOT <= C
LE,C,C,LABEL	J,LABEL	IF C <= C
GE,C,C,LABEL	DELETE	IF C NOT >= C
GE,C,C,LABEL	J,LABEL	IF C >= C
EQ,C,C,LABEL	DELETE	IF C != C
EQ,C,C,LABEL	J,LABEL	IF C = C
NE,C,C,LABEL	DELETE	IF C = C
NE,C,C,LABEL	J,LABEL	IF C != C
AND,C,C,T	SET,CON,T	CON = C AND C
OR,C,C,T	SET,CON,T	CON = C OR C
XOR,C,C,T	SET,CON,T	CON = C XOR C
COMP,C,T	SET,CON,T	CON = COMP(C)
Algebraic Simplification		
+,0,VAR,T	DELETE	IF VAR = T
+,0,VAR,T	SET,VAR,T	IF VAR != T
+,VAR,0,T	DELETE	IF VAR = T
+,VAR,0,T	SET,VAR,T	IF VAR != T
+,C,C,T	SET,CON,T	CON = C+C
-,0,VAR,T	-,VAR,T	
-,VAR,0,T	DELETE	IF VAR = T
-,VAR,0,T	SET,VAR,T	IF VAR != T
-,VAR,VAR,T	SET,0,T	IF VAR = VAR
-,C,C,T	SET,CON,T	CON = C-C
*,0,VAR,T	SET,0,T	
*,VAR,0,T	SET,0,T	
*,-1,VAR,T	-,VAR,T	
*,VAR,-1,T	-,VAR,T	
*,1,VAR,T	DELETE	IF VAR = T
*,1,VAR,T	SET,VAR,T	IF VAR != T
*,VAR,1,T	DELETE	IF VAR = T
*,VAR,1,T	SET,VAR,T	IF VAR != T
*,C,C,T	SET,CON,T	CON = C*C
/,VAR,1,T	DELETE	IF VAR = T
/,VAR,1,T	SET,VAR,T	IF VAR != T
/,VAR,-1,T	-,VAR,T	
/,0,VAR,T	SET,0,T	
/,VAR,VAR,T	SET,1,T	IF VAR = VAR
/,C,C,T	SET,CON,T	CON = C/C
/,VAR,CON,T	*,VAR,INV,T	INV = 1/CON
SET,VAR,T	DELETE	IF VAR = T

Optimal Commuting

LT,VAR,CON,LABEL	GT,CON,VAR,LABEL
GT,VAR,CON,LABEL	LT,CON,VAR,LABEL
LE,VAR,CON,LABEL	GE,CON,VAR,LABEL
GE,VAR,CON,LABEL	LE,CON,VAR,LABEL
EQ,VAR,CON,LABEL	EQ,CON,VAR,LABEL
NE,VAR,CON,LABEL	NE,CON,VAR,LABEL

Logical simplification takes advantage of compile time comparison and compile time logical operations while algebraic simplification utilizes compile time computations.

Although optimal commuting performed here may not apply to some machines other than the 476CX microprocessor, the commuting achieved is still machine independent in the sense that the resultant quads are equivalent to the originals.

The sequences of code shown below illustrate the optimized code generated by the 476CX after optimal commuting is performed.

EQ,V.01,I.05,G5		EQ,I.05,V.01,G5
ADR	XYZ	LET A=5
LET	A=RAM	ADR XYZ
ADR	%CON+0	TEST A+NOT RAM
TEST	A+NOT RAM	IF -1,%L5
IF	-1,%L5	

During code generation, one less ADR instruction is generated.

Other reasons why optimal commuting was included in pass 2 with the other pinhole optimizations include the following:

1. It was a one-quad transformation, the requirement for a pinhole optimization.

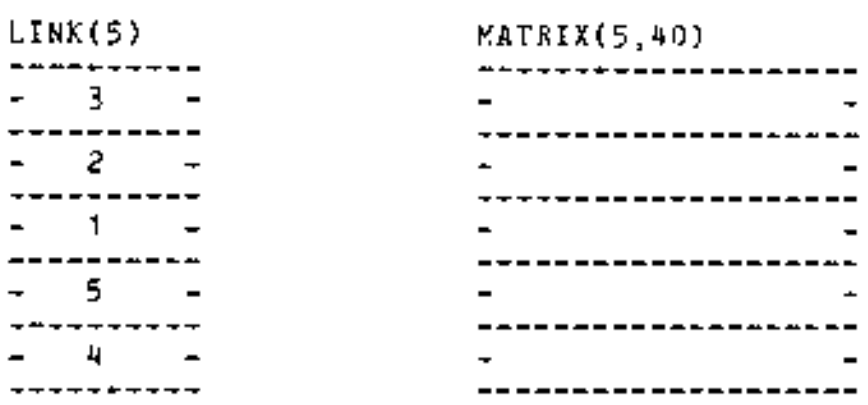
2. This optimization required no machine dependent descriptors, such as register or RAM descriptors that are needed in pass 3.

3. This optimization was easier to implement in pass 2.

The pinhole optimization in which divison by a constant is changed to multiplication by its inverse may also be considered a borderline machine independent optimization. The justification for this transformation is that during code generation, division involves a call to a single or double precision external routine, since the 476 possesses no hardware divide instruction. Less overhead is involved if a harware multiplication is performed.

Pinhole optimizations are performed twice during pass 2. This optimization is accomplished directly after the quads are read in and directly before the quads are written out. The rationale behind this is that constant folding may result in a quad that can undergo further pinhole optimizations.

All optimizations in pass 2 are performed within a five quad window, which is a buffer that possesses the capability of observing five quads at a time. The data structure of the window is shown below:



The link array contains a pointer to each quad in the matrix array. Deletions are performed by updating the pointers and reading a new record into the quad that was just deleted.

The data structure used for the five quad buffer is essentially an open ended singly linked list of quads. The advantage of using this scheme is that deletion of any quad may be performed quickly and easily.

During pinhole optimization, the one quad transformation is performed on the top quad of the window.

The remainder of this section involves a brief description of the peephole optimizations performed during pass 2. Examples are provided to show a sequence of quads before and after optimizaton.

TEMP COMPRESSION

A typical block of quads produced by pass 1 is shown below:

```
*,V.05,V.22,T.01
*,V.06,V.13,T.02
+,T.02,T.01,T.03
SET,T.03,V.03
```

However, in the third quad, neither temporary 2 or temporary 1

will ever be used again. Therefore, the result of the addition operation in the third quad can be assigned to temporary 1.

```
+,T.02,T.01,T.03        --->    +,T.02,T.01,T.01
```

Since temporary 3 is used in the fourth quad, it also must be changed to temporary 1 to reflect the initial change to quad 3. Temp compression will transform the above block of quads into the following block:

```
*,V.05,V.22,T.01
*,V.06,V.13,T.02
+,T.02,T.01,T.01
SET,T.01,V.03
```

Temp compression reduces the number of temporaries, which enables the code generator to allocate the minimum number of temporary locations for a SYS-PROC. Moreover, more efficient assembly code is generated. This optimiztion is performed on the first and second quad of the window.

TEMP COMMUTING

```
*,V.05,V.22,T.01
+,R.05,T.01,T.01
```

Temp commuting will commute the second quad as follows:

```
*,V.05,V.22,T.01
+,T.01,R.05,T.01
```

When code is generated for a given quad, the first operand is loaded into register A (single precision) or AB (double precision) and the second operand is addressed. Since the result of the arithmetic operation of the first quad is left in register A or AB, then if the third operand of the first quad equals the first operand of the second quad, there is a next use in regiser A and a store is not necessary. The temp commuting attempts to create this next use.

CONSTANT FOLDING

```
SET A TO 2*276/92*B
```

The above equation would generated the following set of quads:

```
*,I.02,I.276,T.01
/,T.01,I.92,T.02
*,T.02,V.07,T.03
SET,T.03,V.37
```

It is clear that the first two quads could be evaluated at compile time. The compiler will perform the multiplication and

division for the first two quads and produce the following resulting quads:

```
*,I.06,V.07,T.03
SET,T.03,V.37
```

Constant folding is performed by using a two dimensional array called Tmptbl. Each column of Tmptbl corresponds to a temporary number. The first row corresponds to the type of constant that the temporary in the table presently contains, which is either real or integer. The second row contains the value of the constant. When a source statement is encountered, all entries in Tmptbl are zeroed out. A zero in Tmptbl(1,n) means that the value of the temporary is unknown. If a quad is encountered in which a temp is assigned a constant, then the table is updated with the constants's type and value. If the quad is an arithmetic operator and a temporary in the quad corresponds to a temporary in Tmptbl, then the substitution is made. If the substitution results in both operands being constants, then the computation is performed at compile time, the table is updated, and the quad is deleted.

Tmptbl is initialized after each source statement because the parser generates new temporaries starting with T1 after each source statement.

The parser also tends to generate quads that can be folded when it generates quads for subscripting during array referencing.

If a quad encounters a division by zero, then the quad is left unfolded and pass 3 detects the error and prints out an error message.

GOTO TO GOTO ELIMINATION

The parser generates many goto to goto's during pass 1. Pass 2 will change a quad to branch directly to its destination and delete unnecessary labels that are no longer needed. For example:

```
        GT,V.03,V.08,G.01
G.01:
        J,G.05
.
.
G.05:
        J,G.12
.
.
G.12:
        +,V.12,I.01,T.01
```

WILL BE TRANSFORMED TO

```
GT,V.03,V.08,G.12
.
.
.
G.12:
        +,V.12,I.01,T.01
```

The elimination of goto's to goto's is accomplished by the use of a goto to goto table that is constructed during pass 1. The table is an array Eql(2,200). The first row is a compiler generated label and the second is its branch destination. The following is a goto to goto table taken from a test case:

```
004     002
001     003
010     006
006     007
017     016
015     013
023     021
020     011
021     022
022     011
013     026
032     024
042     037
037     034

ROW1    ROW2
```

At the beginning of pass 2, complete transitive closure is preformed on the goto to goto table, which results in the following table:

```
004     002
001     003
010     007
006     007
017     016
015     026
023     011
020     011
021     011
022     011
013     026
032     024
042     034
037     034

ROW1    ROW2
```

Pass 2 uses the transformed table to perform the optimizations. If a compiler generated label quad is encountered and that label matches an entry on the left hand side of the table, then that label quad is deleted. If a branch quad is found and its destination matches an entry on the left hand side of the table, then that destination is changed to the corresponding right hand entry.

ELIMINATION OF INACCESSIBLE CODE

The compiler will delete those quads which are inaccessible. The algorithm used is as follows. If the current quad is an unconditional branch and the next quad is not a user or compiler generated label, then the next quad is considered inaccessible and is deleted. For example:

```
J,G4
J,G5
```

The second quad will be deleted.

DIRECT TRANSFERS

The elimination of goto's to goto's sometimes produces the following bit of useless code:

```
        J,G5
G5:
```

For this optimization, the jump quad is deleted.

EXIT SWITCHING

An example of a recurring three quad pattern is as follows:

```
        GT,V.01,V.02,G4
        J,G5
G4:
```

This pattern can be optimized by changing the first quad to branch to the false exit and deleting the second quad as shown:

```
        LE,V.01,V.02,G5
G4:
```

PASS 3

The CMS-2 subset compiler was originally designed to be a three pass compiler with all of the code generation occurring in the third pass. However, a very serious problem arouse with branch instructions using this structure. The compiler initially generated short branch instructions using the 'IF' mnemonic, because this was the most efficient method of executing a conditional branch. However, when a program with an extremely large loop was compiled and assembled, the compiler would cite no errors but the assembler would produce a warning that the branch instruction cannot reach its destination. Any branch executed by the 'IF' mnemonic can only reach 128 words away while the GOTO mnemonic is able to reach anywhere within the assembly module. However, more efficient code could be generated using the 'IF' rather than the goto for any branch instruction. It was decided that pass 3 would not generate code for the nine branch intermediate language types (iltypes). If a branch iltype was encountered, the intermediate language record would be passed through as the follows:

FORMAT 1

OPCODE	ILTYPE	LENGTH	IL ARGUMENTS

The meaning of the elements of this record is as follows:

OPCODE - Initialized in this record as a numeric key value for IL. This means that the record contains intermediate language arguments instead of generated assembly language.

ILTYPE - Type of intermediate language (ILJ, ILJEQ, ETC.)

LENGTH - Number of IL arguments that follow the iltype

IL ARGUMENTS - Intermediate language arguments

The structure of the record containing the generated assembly language is as follows:

FORMAT 2

OPCODE	LENGTH	ASSEMBLY LANGUAGE

OPCODE - The numeric enumeration value for the assembly language opcode

LENGTH - The number of characters for the remainder of the record

ASSEMBLY LANGUAGE - Generated assembly language

Therefore, in pass 3 a program counter is incremented each time a line of executable assembly code is generated. If a branch iltype is encountered, the PC is incremented by the amount for a long branch. The branch iltype is then written out to pass 4 using Format 1. Further details on how this optimization is performed will be discussed in the pass 4 section.

LISTING GENERATION/ERROR CHECKING

The first operation that pass 3 performs as the optimized quads are read in is an operand validity check. All the arguments of the quads are checked to determine if the arguments are of the correct type. Pointers are checked to see if they are within the proper range. If any arguments of a quad are in error, Internal Compiler Error Number 11 is printed out to the listing and the terminal. Sometimes user compiler errors cause the parser to produce invalid arguments in quads. In this situation, no internal compiler error is printed and the quad is changed to an ILKILL in pass 3. In other words, no code is generated for this quad.

Pass 3 also generates a listing for the user while generating an assembly language file. The listing contains each CMS-2 statement. If an error occurs, it is written under the CMS-2 statement that is in error. The majority of the errors are detected in the first pass and are passed through to the second and third pass as an intermediate language quad called ILERR. Pass 3 performs its own error checking on the cases where there is division by zero or if a real argument is contained in a Boolean built-in funtion.

One of the problems in pass 3 is the analysis of multiple types of information. The input to this pass includes error iltypes, CMS-2 source statements, and quads. One problem that arose was that warnings would ocassionaly occur between two quads. When this occurred, next use checking resulted with erroneous results because it was performed between the present and and the error quad. The solution to the problem is as follows. If the present quad is a CMS-2 source statement and the next quad is an error, then the error message is generated in the case statement construct of the Pass 3 subroutine. If the present quad is not ILSRC and the next quad is ILERR, then the error message is generated early in the input routine that reads the quads in (Inputb). The next quad is then read into a buffer, thereby overwriting the error quad. The process is repeated until the next quad is not an error quad so that multiple error quads

grouped together will be written out early.

DIRECT CODE

In CMS-2, the user has the option of inserting in line assembly language code in a SYS-PROC. This code is known as direct code. The compiler merely copies this code to the fourth pass untouched. No optimizations are made upon direct code. It is assumed that when the user writes direct code, it is already efficient code.

CODE GENERATION/STORAGE ALLOCATION

The third and fourth passes use a routine called GEN to generate assembly code. Pass is a global variable that records which pass the compiler is currently executing. If Pass equals three, then GEN writes its assembly language to NNN.IL3. If Pass equals four, then the assembly text is written to the assembly language file. Pass 3 uses a routine called OUTIL to output any intermediate language record to pass 4 according to format 1.

Input to pass 3 is buffered in order to perceive next use information. The current quad is contained in an array called Ildata while the next quad is located in Ildatb. Two types of next use checking will be defined here for future reference. One type of next use will be termed next use in register A. An example of this is given below:

```
+,V.01,V.02,T.01
+,T.01,I.05,T.02
```

In this case, the temporary of the first quad has a next use as the first argument as the next quad. Since the first argument of any quad is always loaded into register A or AB pair, then the result of this first quad has a future use in register A.

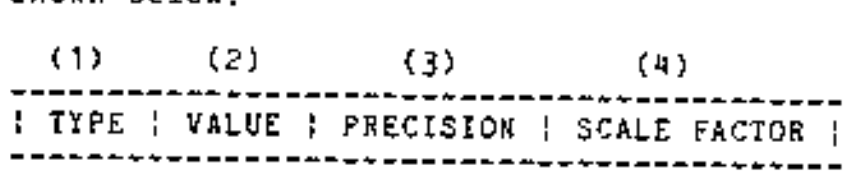
The second type of next use is termed next use in RAM, as shown below:

```
+,V.01,V.02,T.01
+,I.05,T.01,T.02
```

In this case, the temporary of the first quad has a next use as the second operand of the next quad. Because the second argument of any quad is always addressed as a RAM location, the result of the first quad has a future use in RAM.

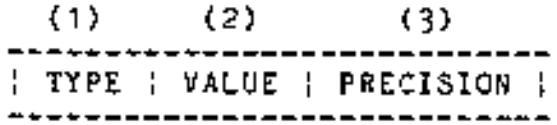
Pass 3 then generates code for the inputted quad and concurrently performs machine dependent optimizations. To make such optimizations possible, it is necessary to monitor the compile time contents of the registers A and B and the RAM location currently being addressed. Moreover, the precision and scale factor of temporaries are also dynamically changing. The

solution to these problems is to maintain a descriptor for the register pair AB and RAM. The register descriptor simulates what is contained in the register pair AB at compile time while the RAM descriptor keeps track of what memory location RAM currently points. The format of the register descriptor is shown below:



The register descriptor is a four element array called Rega. Type and value indicate what constant, temporary, or variable is currently residing in register A. The precision attribute can take on two values: 1 or 2. If precision equals 1 then the item described by type and value is residing in register A. If precision equals 2 then the item described by the first two attributes is contained in register pair AB. The precision attribute enables the usage of one descriptor to record the destination of both single and double precision operations. The fourth attribute is the current scale factor of the result in register A or AB.

The structure of the RAM descriptor is as follows:



The type and value attributes indicate the RAM location to which register 0 currently points. If the third attribute is one, then register 0 points to a single word. If precision equals two then 0 points to a double word in RAM.

Whenever a variable or temporary is stored, the register descriptor is updated. If the item being stored is a temporary, then the current precision and scale factor of the temporary is updated in Tmptbl.

The third pass also has the responsibilty of allocating storage for temporaries and constants. Only those constants that appear as the second operand in a binary operation are allocated in storage. Constants that appear as the first operand in a binary operation are loaded into register A or AB pair using a LET instruction.

Whenever a memory location needs to be addressed, a routine called ARAM is called. This routine checks whether the item to be addressed is a temporary, variable, or constant. If the argument is a temporary, then a search is made through the temp

queue to check if the temporary is already there. If the item is already there, then its displacement is returned. If the item is not already there, then the temporary is enqueued. The same line of logic is followed for the allocaton of constants. If the argument is a variable then the name of the variable is extracted from the symbol table for use as the argument of the ADR instruction.

The temp queue is an array that has three rows and 250 columns. Each column corresponds to a new allocated temporary. Row one is the temp number, row two is the temp's precision, and row three is the temp's displacement. Two temporaries with the same temp number but with different precisions appear as two distinct entities in the queue. They are located in different positions in the queue and possess different displacements. The structure of the constant queue is the same except that the first row contains the constant's value.

Any identifier that is generated by the compiler is preceeded by a per cent sign. Compiler generated labels are in the form:

%Ln	where n is an integer that is >= 1

A compiler reference to a temporary appears as:

%Tmp+n	where n = a displacement

A compiler reference to a constant allocated as a RAM location appears as:

%Con+n	where = a displacement

Double word variables, temps, and constants are stored in memory backwards (ie., low order followed by high order). The reason for this order is due to the observation that more optimal double precision code can be generated.

To perform single precision and double precision division, the compiler calls two external routines, %UDIV1 and %UDIV2 respectively.

The compiler is able to generate code from a quad in which both operands are single precision, both operands are double precision, or if there is a mixture of single and double precision. If both operands are single precision, then the code generated is as follows:

1. Load the first operand into register A (if necessary)
2. Address the second operand as a RAM location
3. Perform the operation (single precision)
4. Store result left in register A to a single word temp (if necessary).

If at least one operand is double precision, then code is generated according to the following algorithm:

1. Load the first operand into register pair AB (if necessary)
2. Address the second operand as a RAM location
3. Perform the operation (double precision)
4. Store result left in register pair AB to a double word temp (if necessary).

If CMODE is specified by the user as being DOUBLE, then under all conditions the operations of addition, subtraction, multiplication, and division are performed according to the double precision sequence.

During a function return, if the function returns a single precision result, then that result is loaded into register A. Else, the result is loaded into register pair AB.

The 476CX microprocessor contains no instructions for floating point operations; all operations are done in fixed point mode. Therefore, if a variable is declared in CMS-2 as integer or arithmetic, then a scale factor is associated with that variable. For example, if a variable is said to have a scale factor of 5, then the binary point lies between the 5th and 6th bit, as shown:

```
---------------------------------------
-0-1-2-3-4-5-6-7-8-9-10-11-12-13-14-15-
---------------------------------------
            $
```

A CMS-2 user determines the position of the binary point by specifying in the variable declaration the number of places that follow the binary point.

When any non-Boolean operation is encountered, scaling must be performed. The compiler increases the scale factor of a variable by generating shift-right instructions and decreases the scale factor by producing shift-left instructions. The following rules are used to perform scaling:

1. If the iltype is ILJEQ,ILJNE,ILJLT,ILJLE,ILJGT,ILJGE, ILADD, or ILSUB, then find the operand with the lower scale factor. Increase its scale factor until it matches the other operand. The resultant scale factor is the scale of the higher operand.

2. For ILMUL, the resultant scale factor is the sum of the scale factors of the operands.

3. For ILDIV, the resultant scale factor is the difference of the scale factors of the operands.

4. For ILABS, ILNEG, OR ILSET, scale the operand according to the destinaton.

5. For ILRETF, scale the returned value according to the scale of the function.

6. For array referencing, scale the item being transferred according to the scale of the destination array. Also, scale the subscript to a scale of 7 for ILIPUT, ILAPUT, ILAGET,or ILIGET. Scaling must be performed here because the subscript of an array is always a single precision integer. Moreover, the scale factor of a double precision integer is always 15. Therefore, the subscript of ILTOAD is scaled to 15.

COMMENT GENERATION

One of the useful functions of pass 3 is the automatic generation of comments. There are two types of comments written to the assembly language file. The first type is generated by an iltype called ILSRC, which contains the text of each CMS-2 statement. These source statements are identical to the ones that appear in the listing file except that they are prefixed by a semicolen, which begins a comment. The other comments are those that the systems programmer generates himself by a call to the GEN routine. Presently, one comment is generated to denote the entry and return from a procedure or function. For example, the following comments are contained in a procedure for an assembly language module:

```
;---------ENTRY TO PROCEDURE

          ASSEMBLY LANGUAGE STATEMENTS...

;---------RETURN FROM PROCEDURE
```

If the block is a function, then 'function' is substituted in the generated comment.

Another comment that is generated to aid the user is the following:

```
;---------LONG BRANCH
```

This comment is generated whenever a long branch is generated by the compiler. The appropriate programmer action is to divide the procedures and functions into smaller blocks of code.

MACHINE DEPENDENT OPTIMIZATIONS

An extensive number of machine dependent optimizations are

performed during pass 3. These optimizations are best illustrated by selected examples. Sequences of code are provided comparing unoptimized code vs. optimized code. The examples given use single precision code in order to allow a clearer comparision but it must be understood that the savings are greater for double precision code. Some optimizations apply to both single and double precision code; if the optimization refers to only one type of precision a comment will appear.

Another important note is that whenever an ADR instruction is saved, two words are saved rather than one because each ADR instruction is expanded into two instructions by the assembler as shown:

```
        ADR  XYZ                 is expanded into

        ADRU XYZ
        ADRL XYZ
```

In the expanded sequence, the first instruction places the 8 most significant bits of the address of XYZ into the upper half of register 0 while the second instruction places the 8 lower bits into the lower half of register 0.

The following summarizes all of the machine dependent optimizations of the CMS-2 subset compiler.

1. Addition of one to a variable or temporary

   +,V.01,I.01,T.01

UNOPTIMIZED CODE	OPTIMIZED CODE
ADR XYZ	ADR XYZ
LET A=RAM	LET A=RAM
ADR %CON+0	LET A=A+1
LET A=A+RAM	(STORE IF NECESSARY)
(STORE IF NECESSARY)	

2. Subtraction of one from a variable or temporary

   -,V.01,I.01,T.01

ADR XYZ	ADR XYZ
LET A=RAM	LET A=RAM
ADR %CON+0	LET A=A-1
LET A=A-RAM	(STORE IF NECESSARY)
(STORE IF NECESSARY)	

3. Loading a variable, temporary, or constant into register A only if the register does not already contain the item.

4. Addressing a RAM location only if register 0 does not already point to it.

5. Storing the result of an operation into a temporary only if there is no immediate next use. For example:

   *,V.01,V.02,T.01
   +,T.01,I.05,T.02

ADR XYZ	ADR XYZ
LET A=RAM	LET A=RAM
ADR ABC	ADR ABC
LET A=A+RAM	LET A=A+RAM
ADR %TMP+0	ADR %CON+0
LET RAM=A	LET A=A+RAM
ADR %TMP+0	
LET A=RAM	
ADR %CON+0	
LET A=A+RAM	

6. For a binary operation, use LETI when both operands of the quad are variables and they are allocated next to each other in storage.

```
+,V.01,V.08,T.02

ADR XYZ                                  ADR XYZ
LET A=RAM                                LETI A=RAM
ADR ABC                                  LET A=A+RAM
LET A=A+RAM
```

In the optimized set of 476 instructions, the ADR instruction directs register 0 to point to the RAM location containing XYZ. The second instruction (LETI) performs the following operations:

a. The contents of register A is loaded with XYZ. Register 0 is incremented to point to the second operand (both operands are juxtaposed).

The third instruction adds the contents of register A to the second operand and stores the result into A.

As a result of this optimization, one ADR instruction is saved.

This optimization is also performed on a SET quad, as shown:

```
SET,V.01,V.02                (ASSUME V.01 IS NEXT TO V.02)

ADR XYZ                                  ADR XYZ
LET A=RAM                                LETI A=RAM
ADR ABC                                  LET  RAM=A
LET RAM=A
```

7. If the present quad is addition or subtraction and there is a next use in RAM, then store the result of the operation into RAM rather than into register A.

```
+,V.01,T.08,T.08
-,V.05,T.08,T.03

ADR XYZ                                  ADR XYZ
LET A=RAM                                LET A=RAM
ADR $TMP+7                               ADR $TMP+7
LET A=A+RAM                              LET RAM=A+RAM
LET RAM=A                                ADR ABC
ADR ABC                                  LET A=RAM
LET A=RAM                                ADR $TMP+7
ADR $TMP+7                               LET A=A-RAM
LET A=A-RAM
```

This optimiztion eliminates an unnecessary store operation.

If the present quad is addition or subtraction and the next quad is a ILSET and there is a next use in RAM, store the

result into RAM rather than register A. For example:

```
+,V.01,V.02,T.04
SET,T.04,V.02
```

```
ADR XYZ                          ADR XYZ
LET A=RAM                        LET A=RAM
ADR ABC                          ADR ABC
LET A=A+RAM                      LET RAM=A+RAM
LET RAM=A
```

When a user writes CMS-2 statements in which a variable is incremented or decremented by one and stored into the same variable, a next use in RAM is also produced, as shown.

```
SET XYX TO XYZ+1    GENERATES   +,V.01,I.01,T.01
                                SET,T.01,V.01
```

```
ADR XYZ                          ADR XYZ
LET A=RAM                        LET A=RAM
LET A=A+1                        LET RAM=A+1
LET RAM=A
```

In the optimizations in which there is a sequence of addition, set, or subtraction quads, then after the result has been stored into RAM, no code is generated for the set quad. This is done by generating the optimal code for the present quad (addition or subtraction) and changing the next quad (ILSET) to ILKILL.

8. Generate optimal code for the squaring of a temporary or variable. This optimization applies only to double precision operations.

```
*,V.01,V.01,T.01
```

```
ADR  XYZ                         ADR  XYZ
LETI B=RAM                       LETI B=RAM
LET  A=RAM                       LET  A=RAM
ADR  XYZ                         LET  B=B
LETI B=RAM                       LET  A=A
LET  A=RAM                       MUL
MUL
```

The multiply instruction works by multiplying the contents of the register pair AB by A'B'. In the squaring case, it is more efficient to retrieve the second operand from registers A and B rather than from RAM.

9. If a quad is encountered that is a multiplication by a power of two, then no assembly code is generated. Instead, increase the scale factor containing the result by the power of two.

10. If a quad is encountered that is a division by a power of two, do not generate any assembly code. Instead, decrease the scale factor of the temporary containing the result by the power of two.

The algorithm in which optimizations numbers 9 and 10 are implemented is as follows. Suppose a temporary is multiplied by two in a quad as shown below.

*,T.01,I.02,T.01

The temp table is updated with the temp's resulting precision and scale. In this case, the scale of T.01 is incremented by one.

Suppose the quad was the multiplication of a variable by a variable and stored in temporary one.

*,V.01,I.02,T.01

In this case, one possible solution would be to increase the scale factor of the variable by one in the symbol table. However, this cannot be done because the scale factors of the variables must remain constant for this is the requirement of fixed point operations. What is done instead is that the scale factor of temp 1 in the temp table is incremented by one as before. Next the pointer to the variable is also entered into the temp table for temp one. Whenever temp one is referenced as a future quad within a basic block, the name of the variable will be used but the scale of the temporary is used. What has occured here is that the scale of the variable has been updated to repesent the multiplication without resorting to changing its value in the symbol table.

If the destination of the multiplication by a power of two operation is the same variable, then the only scaling performed is the variable assignment. This scaling is performed separately and is done for a ILSET quad. However, for the multiplication or divison quad, no scaling code is generated in this case.

11. For addition, subtraction, and comparison quads, scaling is performed by increasing the scale factor of the operand with the lower scale. If one of the operands is a constant, then there is no problem because the scaling can be done at compile time. However, if both operands are variables, then scaling is performed by loading the operand with the lower scale factor into register A or AB and shifting right until its scale has increased sufficiently.

The most optimal case is the situation in which the scale

factor of the first operand is less than the scale factor of the second. If the scale factor of the first operand is greater than the scale factor of the second, then the following type of code is generated:

1. Save the contents of regiser A or AB pair in register RR.

2. Load the second operand into register A or AB. Perform scaling by shifting right.

3. Store the contents of register A or AB back into the second operand (a RAM location).

4. Load the first operand from RR back into register A or AB.

Therefore, if both operands are variables or temporaries and the first operand's scale factor exceeds the second and the iltype is ILADD or ILJNE or ILJEQ, then the operands are commuted.

12. When the precision of the two operands is unequal, then it is not optimal when the precision of the first operand is double and the second is single. If this is the case, the following type of code must be generated:

1. Save the contents of register AB into register RR.

2. Load the contents of the the second single precision operand into register A.

3. Extend the operand to double precision by loading a zero into register B.

4. Allocate a new temporary on the temp queue.

5. Store the contents of AB into the new temporary.

6. Generate an ADR instruction so that register 0 now points to the new temporary.

The above algorithm results with the extension of single precision to double precision for the second operand by allocating a new temporary. Since this is not efficient code, if the iltype is ILADD, ILMUL, ILOR, ILAND, ILXOR, ILEQV, ILJEQ, or ILJNE, then the operands are commuted.

13. If one operand is a constant and the other operand is a variable and scaling needs to be done, then perform the scaling at compile time.

14. The generation of a shift instruction will be optimized. If the number of right arithmetic shift instructions required is greater than three, then a multiply instruction will be generated to perform the shift. For the other types of shifts, if the number of shift instructions is greater or equal to eight, then eight less shift instructions are generated by performing register swapping before the shifts.

15. Whenever pass 3 encounters a compiler generated label, assembly code is generated and the register and RAM descriptors are initialized as empty, since the contents of the descriptors are unknown before the branch to the label is executed. If pass 3 encounters a compiler label, then a check is made in the label reference array to determine whether this label is the destination of any branch quad. If not, then no code is generated for this quad and the register and RAM descriptors are left as is.

PASS 4

As mentioned earlier, pass 3 generates codes for all type of quads except the following:

1. Branches
2. Subroutine calls
3. Entry to procedure or function
4. Return from a procedure or function

The code for the above quads is generated during pass 4. Assembly language statements are merely written to the ASM file unchanged.

The sections that follow describe the method of generating code for the forementioned quads.

BRANCHES/SUBROUTINE CALLS

Two tables that are built by pass 3 but utilized by pass 4 include the label and goto tables. Their formats are shown below:

LABEL TABLE RECORD

TYPE
VALUE
PC

BRANCH TABLE RECORD

TYPE
VALUE
ILTYPE
INDIC
PC
PREC

Type indicates whether the branch is a user or compiler generated transfer. Indic indicates whether the transfer is long or short while Prec indicates the precision of the operation (single or double).

Whenever a label is encountered during pass 3, an entry is made into the label table. When the current iltype is a branch or call, an entry is made into the goto table. Indic is initialized as two, which represents a long branch.

At the beginning of pass 4, these two tables are run through an optimizing process. When a transfer iltype is recognized in this pass, the record containing the branch's label destination is found in the label table. The difference between the PC of the label and the PC of the branch is calculated. If the difference is greater than -127 and less than or equal to 128, then Indic is changed to 1, indicating that this is now a short branch. The PC of each succeeding branch record in the branch table is decreased by the difference between a long and short branch. Next, the PC of each label occurring after the branch in question is decremented by the difference between a long and short branch. The algorithm is repeated until no more branches can be converted to short branches.

The table below lists the number of words generated by long branches and short branches. Both single and double precision cases are considered. With iltypes such as ILJ,ILJF,ILJT,and ILCALL, precision has no real meaning. For the sake of clarity, they are placed in the single precision column.

	SHORT BRANCH		LONG BRANCH	
	Single	Double	Single	Double
ILJ	1		2	
ILJF	2		5	
ILJT	3		4	
ILJEQ	2	5	5	8
ILJNE	3	4	6	8
ILJLT	2	5	3	6
ILJGE	2	5	3	6
ILJLE	2	5	3	6
ILJGT	2	5	3	6
ILCALL	1		2	

The generation of assembly language for branch iltypes entails the usage of both relative and absolute branching. Absolute branching is used to jump to the final destination while relative branching is used to jump within the block of assembly language instructions generated for a given quad. For example, the following is used to generate code for a single precision long branch for ILJLT. Assume that the first argument is already in A and register 0 already points to the second operand.

```
LET A=A-RAM
IF A LT 0,$+2
IF TRUE,$+3
GOTO LABEL
```

The dollar sign represents the instruction location tag. The advantage of using relative addressing is that the code

generated is position independent.

ENTRY/RETURN PROTOCOL

When an assembly module is called using the CALL or IFS mnemonic, the program counter is stored in register SC. This method works fine until the procedure that was called contains another call instruction. If multiple subroutine calls occur, then the original return address found in SC is destroyed.

Each CMS-2 SYS-PROC generates an assembly language module and each function or procedure generates a label entry point and a return instruction. Since there is no concept of a main procedure in CMS-2, it is assumed that all procedures or functions have equal probability of being called by another procedure or function. Given a procedure, because it is not known that this procedure is the main procedure (this is determined at link time), it is assumed that the given procedure was just called by some 'main' procedure. Therefore, if the given procedure contains a CALL or IFS mnemonic, the long entry and return protocols are generated. Upon entry, the return address found in SC is stored into a RAM location. Upon return, the return address in RAM is transferred to register RR. RR is a return register that can be explicitly loaded. Exection of the RRR instruction transfers the contents of RR to the program counter (CC).

Therefore, there exit two types of entry-return protocols: long and short. If there are no subroutine or function calls in a procedure or function, the short subroutine protocol is executed. Otherwise, the long protocol is performed.

Given a procedure called PROC, the following examples represent the long and short protocols respectively.

```
PROC    EQU     $
        .
        .
RETURN
```

```
PROC    EQU     $
        DSECT
%RA1    RES     2
        PSECT
        ADR     %RA1
        LETI    RAM=SCU
        LET     RAM=SCL
        .
        .
        ADR     %RA1
        LETI    A=RAM
        LET     B=RAM
        LET     RR=AB
        RRR
```

An inherent part of the long protocol is the allocation of a return address variable. This variable is contained in the DSECT or data section as shown.

CONCLUSIONS

In general, the advantage of using a compiler is to alleviate the difficulty involved with programming a machine. It is not necessary to concern oneself with the contents of registers, RAM locations, and the state of machine flags. When military applications are concerned, fixed point arithmetic is commonplace; therefore, the programmer must pay attention to scaling. This is a rather difficult and tedious task on the assembly language level for large programs. For the CMS-2 compiler, this tedium is lessened by the fact that the compiler performs the scaling for the user through the use of tables and the generation of proper shift instructions. However, in order for the compiler to generate the most efficient code, the user must be concerned with scaling even on the level of CMS-2. It has been noted that when variables in expression can be grouped to produce the minimum number of shift instructions for scaling. For example, the first equation contains variables that are grouped in a nonoptimal way while the variables in the second equation are grouped more efficiently.

```
    1     2  1  -2   1  -2     1 -2   1
SET AP TO DD*AA-DPHI*BB-DTHETA*C-DPSI*D $

    1       -2   1  -2     1 -2   1  2  1
SET AP TO -(DPHI*BB+DTHETA*C+DPSI*D)+DD*AA $
```

The following summarizes how code for each equation is generated:

Equation 1

```
T1 <- DD*AA              3
T2 <- DPHI*BB           -1
T2 <- T1-T2              3      NONOPTIMAL SCALING IS PERFORMED
T4 <- DTHETA*C          -1
T4 <- T2-T4              3      NONOPTIMAL SCALING IS PERFORMED
T6 <- DPSI*D            -1
T6 <- T4-T6              3      NONOPTIMAL SCALING IS PERFORMED
```

Equation 2

```
T1 <- DPHI*BB           -1
T2 <- DTHETA*C          -1
T1 <- T1+T2             -1
T4 <- DPSI*D            -1
T1 <- T4+T1             -1
T6 <- -T1               -1
T7 <- DD*AA              3
T6<- T6+T7               3      OPTIMAL SCALING
```

In general, scaling for additon or subtraction involves scaling the operand with the lower scale factor until it matches the other operand's scale factor. In the first equation, it is compulsory that scaling be performed because the scale factors were not equal. Moreover, each time unoptimal scaling has to be performed because the scale factor of the first operand is greater the second. (see machine dependent optimization 11 for more details). In all three cases, scaling is done for subtraction. For the second equation, optimal scaling is performed for addition. In this equation, more optimal code is generated because the scales are grouped together so that scaling only has to be performed once. Moreover, when scaling is performed, it is optimal because the scale factor of the first operand is less than that of the second.

In general, the following rules generate better code:

1. Group scale factors together so that addition and subtraction operands have equal scale factors.

2. Factor out minus signs in equations. This will change the subtraction operations to addition operations. Addition operations are more desirable because the compiler can commute the operands if a non-optimal scaling situation exists.(if scale factor of the first operand exceeds that of the second.

In the construction of the CSM-2 compiler, the parser and symbol table are generated by the Metatranslator. Whenever a new symbol is hashed into the symbol table, a pointer is set to point to a node containing information about that symbol. However, this node would also point to a previous node allocated in that same list via foward and backward pointers. The added effort to establish the list pointers entails unecessary overhead on the part of the Metatranslator. The first change to the compiler would be to install a new command in the Metalanguage that would allocate a free node without the forward and backward pointers.

The CMS-2 subset compiler already contains intrinsic functions to perform logical and shift operations. These built in functions are simple enough to merely generate 476CX assembly language without having to resort to a subroutine call. In the future, more intrinsic functions will be added in order to extend the power of the compiler. Library routines will be written in CMS-2 in order to perform the common mathematical functions such as cosine, sine, arctangent, hyperbolic sine, and hyperbolic cosine.

Because the CMS-2 compiler will be used for real time applications, it is necessary for the mathematical routines to use a highly efficient algorithm. It would be desirable to choose an algorithm which does not contain any division and

multiplication, for these are usually the most expensive operations on any machine. The Cordic Algorithm is an example of such a method. With this process, the following mathematical functions can be calculated using only two's complement arithmetic and shifting:

LINEAR FUNCTIONS

Multiplication
Division
Decimal to Binary Conversion
Binary to Decimal Conversion

CIRCULAR FUNCTIONS

Cosine
Sine
Arctangent

HYPERBOLIC FUNCTIONS

Hyperbolic Sine
Hyperbolic Cosine
Arc Hyperbolic Tangent
Square Root
Natural logarithm
e raised to a power of X
e raised to a power of -X

The Cordic algorithm requires that the precision of the result be declared in the number of bits. This is easily implemented in CMS-2, since each variable declared contains an implicit binary point. Moreover, CMS-2 possesses a built-in right arithmetic shift instruction that is required by the algorithm.

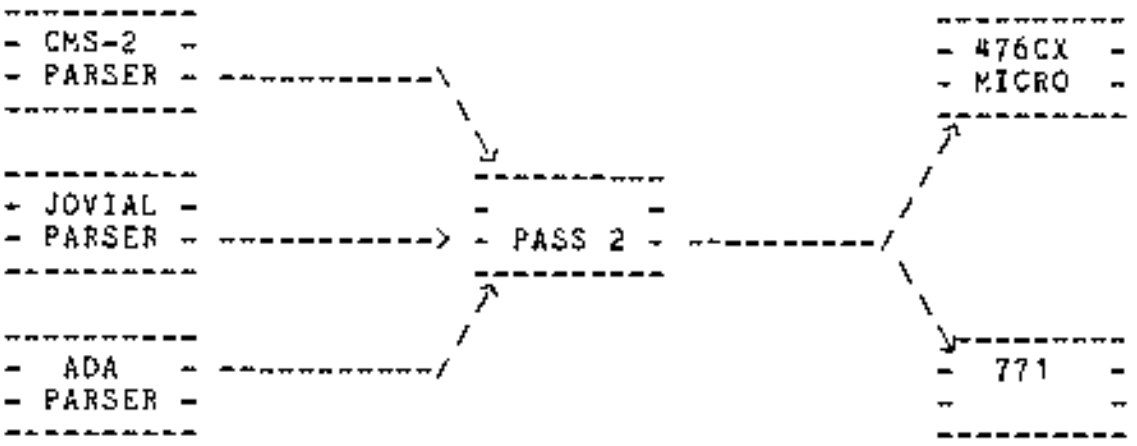
The real advantage of using the Cordic algorithm is its speed. However, the algorithm possesses two disadvantages. One disadvantage is that each function or set of functions contain a specific domain in which the Cordic algorithm will produce correct results. For example, for multiplication and division, the boundary requirements is as follows:

$$2^{**}I <= Z < 2^{**}I+1$$

Z is the multiplier or multiplicand and I is some integer. Another problem is the storage requirement. The circular and hyperbolic functions require a table of constants. The amount of storage (in bits) needed to construct the constant table is N**2*N, where N is the number of bits of precision desired. However, the linear functions do not require a constant table.

THE CMS-2 SUBSET COMPILER

One of the strong points of the compiler is its modular design. Each pass consists of a separate main routine that call support routines. Moreover, each pass is contained in a separate file. For the future, it is possible to expand the present CMS-2 compiler into a multi-compiler that would compile different military languages for different target machines. A diagram of such a system is shown below:



The illustrated system would have the capability of cross-compiling for any of the three military languages to either target machine. Each phase would share a common symbol table and use the same machine independent language. The compiler user would execute a command file that would link together the desired parser and target machine load modules before running the compiler.

REFERENCES

Aho, A. V., and Ullman, J. D. [1977]. Principles of Compiler Design, Addison-Wesley, Reading, Massachusetts.

Badura, J. T. [1980] CMS-2 User's Manual, MCDONNELL DOUGLAS ASTRONAUTICS COMPANY, Huntington Beach, California.

Badura, J. T. [1980] CMS-2 Test Cases, MCDONNELL DOUGLAS ASTRONAUTICS COMPANY, Huntington Beach, California.

Gries, D. [1971]. Compiler Construction For Digital Computers, John Wiley, New York.

Metatranslator User's Manual Version 2. [1979] MCDONNELL DOUGLAS ASTRONAUTICS COMPANY, Huntington Beach, California.

Computer Program Performance Specification for AN/UYK-20 CMS-2M Compilers. [1977] Department of the Navy.

Swim, B. R. [1980] 476CX Assembler Manual, MCDONNELL DOUGLAS ASTRONAUTICS COMPANY, Huntingtom Beach, California.

Volder, J. E. [1959]. "The Cordic Trigonometric Computing Technique," IRE Transactions on Electronic Computers, Vol EC-8, No. 3, 330-335.

Walther, J. S. [1971]. "A Unified Algorithm for Elementary Functions," American Federation of Information Processing Societies, Inc.