

Introduction ... May 18, 2024

The next line is 94 chars long:

0123456789 0123456789 0123456789 0123456789 0123456789 0123456789 0123456789 0123456789 0123456789 01234567890123

Note that other than tabs (8 chars) and carriage returns there are NO page breaks in the document.

To be useful the IMS one needs the source code, a working compiler & the ability to build & run a compiler in test mode

SEE <http://www.jasik.com/CCG.html> for info (URL is CASE Sensitive)

SIJ NOTES appear in a few places in the IMS. They are my personal comments

This file was scanned in with help from Paul McJones in March 2023 and corrected.

The original document was printed on a chain printer & a brief inspection of it will make its faults evident.

The results of it is on CCG_IMS_1978.pdf (5MB) & can be used when errors are suspected in the edited document.

I was the original author/project mgr of the the optimizer along with Martin Herbach and Jean Paulson. The original code was written starting in August 1972 and released in the Fall of 1974 as the code generation and assembly passes of FTN 4.3 on the Scope 3.x & Kronos operating systems. The current version available is FTN 5 which uses the CCG for Optimization levels 1, 2 & 3.

THE CDC 6600 was introduced to the world in September 1964 at a base price of \$6.9 Million.

The 1st 2 machines went to the Livermore & Los Alamos National Labs.

The NYU Graduate school of Math, The Courant Institute of Mathematical Sciences (CIMS) got SN #4.

I was a graduate Student there from 9/1965 - 7/1968 & 7/70 - 7/72.

In July 1968 I left NYC to join the CDC 6000 series development facility in Palo Alto California.

Initially I worked on the operating system. Later I joined the FTN compiler project working on a reference map. In July 1970 I returned to grad school to possibly complete my PhD studies.

I was also working at CIMS as a CDC post sales site analyst assisting users...

Jack Schwartz ran the Comp Sci Dept. Fran Allen & John Cocke of IBM Research worked with Jack on compiler optimization which he was interested in.

In the spring of 1972 CDC invited me to rejoin them in California & develop a global optimizer for FTN which is described in this document. The goal of the project were to do optimizations so as to make the compiler competitive with IBM's FORTRAN H compiler.

I worked on it until around 1980 or so when I left CDC and wandered around Silicon Valley working for different firms until in 1984 I discovered Macintosh computers and wrote programming tools (MacNosy & The Debugger) for Apple programmers as an Independent Developer.

FTN 4.3 was released to CDC 6000 customers in the fall of 1974.

Today's Comp Sci students should note that as FORTRAN was the main language. Our customers were Fortune 500 companies, Universities & Government research organizations that we had a large database of test programs including the notorious "CERNs" which had to be passed.

Our 3 person team spent at least 3 weeks in Minneapolis running on a 7600 getting the CERNs to work in OPT=2. Along the way we picked up a copy of "PDQ", a nuclear reactor code to play with.

This document was produced on a Macintosh computer using OSX 10.14.6 (Mojave) in TextEdit as prior attempts to scan and format the original paper copy of it were too full of scan program mis-translations and way too cumbersome to modify in MS Word.

I have made over 3 passes over the original OCR scan to correct it.

Common errors that OCR scan program made were D -> O or Ø, comma -> *, R -> P, B -> 3 ...

Errors can be sent to me at Steve@jasik.com along with a page # and a line or 2

At this point in time it is available as: a pure text file, an RTDF & a .pdf .
Also, there is a copy of the original scan in case you suspect a mis-translation.

This text file is best viewed at a page width of 94 columns wide in the font "Monaco 9"
Or equivalent fixed width.

Yes I did take advantage of Apple special chars like Delta Δ (option j) in one place

The lines of the form: "[[PAGE BREAK-- end ss-nn" correspondent to the page numbers
in the table of contents. I can use the Find command to move to a page or section

Annals of History of Computing - antidotes

Google "cocke & schwartz compiler book"

a copy the 762 page book:

https://www.softwarepreservation.org/projects/FORTRAN/CockeSchwartz_ProgLanCompilers.pdf

https://en.wikipedia.org/wiki/History_of_compiler_construction

A more modern book is the Aho Ullman Compilers Text

Prefetch Timing example: search the CCG IMS for: "PreFetch T"

1st Plist word & PC(opt) - normalization

The Computers

For those not familiar with the Control Data computers of the 1960's the
wikipedia page does a decent job of describing the machine & it's history:

https://en.wikipedia.org/wiki/CDC_6600.

This page is interesting: <http://www.couperus.org/Albums/CDC6600/>

A more complete history of CDC's machines:

https://en.wikipedia.org/wiki/CDC_Cyber

<https://www.computerhistory.org/brochures/a-c/control-data-corporation-cdc/>

The CDC 6000 & 7000 super computers both had 60 bit words which contained space for
10 six bit characters limiting one to only uppercase characters.

In addition memory was limited to one mega word of 18 bit addresses (131,072)

And there was no Virtual Memory.

In addition the high end machines 6600 & 7600 had multiple arithmetic/logic units
which allowed multiple instructions to be executed in parallel.

There were eight 60 bit X registers,

eight 18 bit index "B registers" and eight 18 bit "A registers"

You loaded a value into an X register by setting an address into A1 - A5 and
stored a value from X6 or X7 by entering an address into the corresponding A-reg.

In addition there was a Scoreboard which allowed the CPU to issue instructions
before the operands to it were in their registers. This way multiple instructions could
be in progress at any given time.

There was also an 8 word instruction stack where one could hold upto 30 15 bit
instructions and one jump back to the top of a loop.

The register design was an idiosyncratic mess with no thought given to difficulties that a compiler writer would face when doing register assignment. The basic problem was the use of 60 bit words & 6 bit characters and the instruction opcodes were limited to 6 bits hence only 64 instructions were possible. If instead 64 bits words with 7 bits for the opcodes would have allowed for 128 opcodes... With that as a solution one could have load & store instructions that worked for all the X registers instead of the design of 8 X registers of which one could only load from memory into X1 - X5 and store into memory from X6 & X7. The given reason for 6 bits and octal numbers instead of 8 bits and Hex was that Seymour Cray didn't have a symbolic assembler and wrote code directly in Octal :-(

The Intermediate Language (IL)

The output of the 1st pass of the compiler was a sequence of language dependent macro calls. The links between the output of one macro would become the inputs to the successor macro call that referenced it. WE called the links R-numbers and they occupy 16 bit fields. RI, RJ & RK are refs to R-numbers. RI denote a result & RJ, RK operands. In the original design the low level instructions that the code generator worked with occupied 3X 60 bit words.

When the macros are expanded into sequences of one or more low level instructions in the code generation passes we generate intermediate R-numbers with the high bit set such as 100001B (Octal). This creates a problem for SQZ and other routines as it would be nice if the R-numbers could have some simple order so as to make looking backward a simple indexing operation instead of having to use a hash table or such.

My solution was two fold.

I added an extra "Link" word to each instruction so that the result of an instruction (RI) was always a multiple of 4.

And wrote a routine RNI (Renummer Instructions) in BDT to bring all the R-numbers into this "canonical" form.

In the original design the low level instructions that the code generator worked with occupied 3X 60 bit words.

PROCEDURE NAMING

The majority of the routines are 3 letters long and are some combination of Verb Adjective, Noun

CCG IMPLEMENTATION LANGUAGE

JAW See Section 4.0

Look for an example on <http://www.jasik.com/CCG.html> (URL is CASE Sensitive)

CDC Vocabulary from the 1960 - 198x

Core -> Memory RAM

Update & Modify were the source code management programs.

References to OLDPL is the file containing the source code and modification history

Source code files were organized into DECKS which contained one or more routines while COMDECKS were the equivalent of modern Include files that were "called" by multiple DECKS.

Keypunches & card readers - After one wrote a computer program on paper, one used a keypunch to write it onto 80 column Hollerith cards which were then fed into a computer file as part of a "job" via a Card Reader.

Remote terminals & Time Sharing were invented on the 6600 around 1968...

PLATO terminals came somewhat later

Miscellany

Tim's Blog - what made the 6600 fast

<https://cpldcpu.wordpress.com/2020/02/14/what-made-the-cdc6600-fast/>

Jim Thornton's book as a .pdf - see pages 12 - 17 - rational for CPU design

<https://archive.computerhistory.org/resources/text/CDC/>

[cdc.6600.thornton.design_of_a_computer_the_control_data_6600.1970.102630394.pdf](https://archive.computerhistory.org/resources/text/CDC/cdc.6600.thornton.design_of_a_computer_the_control_data_6600.1970.102630394.pdf)

---END INTRO---

[[] PAGE BREAK--

TABLE OF CONTENTS

Section	Title	page
1.0	INTRODUCTION	1-1
1.1	OVERVIEW	1-1
1.2	CCG INPUT	1-1
2.0	ENVIRONMENT.....	2-1
2.1	RESOURCE SET (REGISTERS & INSTRUCTIONS)	2-1
2.2	PERT-TIME & INSTRUCTION SCHEDULING	2-5
3.0	GENERAL THEORY OF OPTIMIZATION	3-1
3.1	MEMORY REFS, SEMANTIC INFO & OATA INTERFERENCE	3-2
3.2	REGISTER ASSIGNMENT & DEADLOCK SITUATIONS	3-4
3.3	REFERENCES:.....	3-6
4.0	CCG IMPLEMENTATION LANGUAGE	4-1
4.1	DEBUGGING FACILITIES	4-5
4.2	INTERNAL LANGUAGE DESCRIPTION	4-6
4.2.1	OPROEFS	4-6
4.2.2	CCG ONLY INSTRUCTIONS	4-7
4.2.3	THE SO FIELD & REGISTER SPECIFICATIONS	4-7
4.2.4	PEGISTER REDEFINITION.....	4-8
5.0	TABLE & STRUCTURE DEFINITIONS	5-1
5.1	STRUCTURES.....	5-4
5.2	GENERAL PROGRAM FLOW	5-8
6.0	COMOECKS.....	6-1
7.0	TEXTS	7-1
7.1	CMPLTXT.....	7-1
7.2	CCGTEXT.....	7-1
8.0	CCG DEBUG MOOE PRINT ROUTINES	8-1
8.1	ROUTINES	8-1
8.2	EXTERNALS REFERENCED	8-2
8.3	ROUTINE STRUCTURE AND COOING CONVENTIONS.....	8-2
8.4	CALLING SEQUENCES	8-3

9.0 CGIA - COOE GENERATOR INTERNAL ASSEMBLER	9-1
9.1 GENERAL BLOCK STRUCTURE OF CCG ASSEMBLER	9-1
9.2 INITIALIZATION OF CGIA PROCESSING	9-3
9.3 MAIN CONTROL LOOP (GNIW) & INSTRUCTION PROCESSORS	9-4
9.3.1 PROCESSING OF INITIAL PSEUOO INSTRUCTIONS	9-4
9.3.2 PROCESSING OF MACHINE INSTRUCTIONS	9-5
9.3.3 PROCESSING OF MEMORY REFS & REGISTER SET INSTRUCTIONS	9-6
9.3.4 PROCESSING OF JUMP INSTRUCTIONS.....	9-6
9.3.5 PROCESSING OF STANDARD PSEUOO-INSTRUCTIONS	9-7
9.3.6 PROCESSING OF SLIST MACRO REFERENCES	9-9
9.3.7 END PSEUOO PROCESSING	9-11
9.4 THE OBJECT LISTING	9-12
9.5 LOADER TABLE OUTPUT ROUTINES	9-13
9.6 CGIA DATA STRUCTURES	9-14
9.6.1 CGIA TABLES	9-15
9.6.2 LOCAL VARIABLES	9-17
10.0 MACROS - HOST COMPILER STORAGE MACRO SKELTONS	10-1
10.1 ENTRY POINTS	10-1
10.2 MACPO DEFINITIONS.....	10-1
11.0 CGTM - COOE GENERATOR TABLE MANAGER	11-1
11.1 ENO PROCESSOR ROUTINES	11-1
11.2 WII - WRITE ISSUED INSTRUCTION TO SLIST	11-2
11.3 TABLE MANAGER	11-4
12.0 MIO - OPT«2 MASS STORAGE I/O ROUTINES.....	12-1
13.0 FBV - FORM BIT VECTORS.....	13-1
14.0 GPO - GLOBAL PROGRAM OPTIMIZATION.....	14-1
15.0 GPA - GLOBAL REGISTER ASSIGNMENT.....	15-1
16.0 PROSEQ - SEQUENCE PROCESSING CONTROL	16-1
17.0 MCG - MACHINE COOE GENERATOR	17-1
17.1 JAM	17-3
18.0 SQZ - REMOVE COMMON SUBEXPRESSIONS	18-1
19.0 BDT - FORM DEPENDENCY GRAPH.....	19-1
19.1 RIO - RESET INSTRUCTION ORDER.....	19-2
19.2 RNI - RENUMBER INSTRUCTION R-NUMBERS	19-3
20.0 CCG UTILITIES - ROUTINES TO FORMAT DEBUG OUTPUT	20-1
20.1 CFA - CONTROL FLOW ANALYSIS.....	20-1
20.2 UDT - USE-DEFINITION TABLE PROCESSING	20-4
21.0	21-1
CCG INTERFACE SPECIFICATION APPENDIX.....	21-1
1.0 INTRODUCTION	21-1

1.1	COMPILER COMPONENTS	21-1
1.2	COOE GENERATOR COMPONENTS	21-1
1.3	COOE GENERATING OVERLAY CONTROL	21-4
1.4	STRUCTURE OF THIS DOCUMENT.....	21-5
1.5	NOTATION CONVENTIONS	21-5
1.5.1	Table Description Notation	21-5
1.5.2	NAMING CONVENTIONS	21-8
2.0	THE SYMBOL TABLE & ASSOCIATES	21-11
2.1	SYMBOL TABLES.....	21-11
2.1.1	The Main Symbol Table - SYM	21-11
2.1.1.1	Word 1 (Word A) Format and Usage ..	21-12
2.1.1.2	Word 2 (Word B) Format and Usage ..	21-12
2.1.1.3	Word 3 (Word C) Format and Usage ..	21-14
2.1.2	Abbreviated Symbol Tables	21-15
2.1.3	Symbol Table References	21-15
2.1.4	Predefined Symbols.....	21-16
2.2	ASSOCIATED TABLES.....	21-16
2.2.1	Block Tables - CBT, L3T	21-16
2.2.2	Constant Tables - CVT, CUT	21-17
2.2.3	Variable Dimension Info Table - VDI ..	21-18
2.2.4	Formal Parameter Info Table - FPI ...	21-18
3.0	THE CODE TRANSFORMER.....	21-20
3.1	OPTIMIZATIONS	21-20
3.1.1	Resolution of Memory Refs	21-20
3.1.2	Straight Line Code Optimizations	21-22
3.1.3	Innermost Well-behaved Loop Optimiz..	21-24
3.1.4	Global Optimizations	21-26
3.2	AUXILIARY FUNCTIONS	21-27
3.2.1	Code Transformer Generated Temps.....	21-27
3.2.2	Address Expansion and Substitution ..	21-28
3.2.3	Address Assignment for Labels	21-29
3.2.4	Code Length Computation	21-29
3.2.5	Materialization of Variables & Consts	21-29
3.2.6	Dead Code Error Messages	21-30
4.0	THE BRIDGE.....	21-31
4.1	GENERAL FLOW.....	21-31
4.2	COOE SEQUENCES	21-31
4.3	THE INTERNAL LANGUAGE	21-33
4.3.1	Operand Links (R-Numbers)	21-33
4.3.2	Instruction Types	21-35
4.3.3	Instruction Set	21-36
4.3.4	Instruction Set Rules	21-42
4.3.5	Instruction Formats	21-42
4.4	CONTROL FLOW INFORMATION	21-44
4.5	USE/DEFINITION INFORMATION	21-44
4.6	Packing of ST.s (Function Temporaries) .	21-46
5.0	THE ENO PROCESSOR,.....	21-47
5.1	COMPLETING THE SLIST FILE	21-47
5.1.1	Prologue Code	21-47
5.1.2	Constants & Temporary Storage	21-47

5.1.3 ENO Card	21-47
5.2 COMPLETING THE TABLES	21-48
5.2.1 The Local Block Table	21-48
5.2.2 Word C of the Symbol Table	21-48
5.2.3 Temporary Storage	21-48
5.3 ADJUSTING MEMORY LIMITS.....	21-48
6.0 INTERNAL ASSEMBLER	21-49
6.1 INTRODUCTION	21-49
6.2 ASSEMBLER INPUT	21-49
6.3 SLIST FILE.....	21-50
6.4 MACHINE INSTRUCTIONS	21-50
6.5 SLIST PSEUDO INSTRUCTIONS	21-55
6.5.1 Initial Group	21-55
6.5.2 Declarative Group	21-56
6.5.3 ENO Instruction.....	21-59
6.6 MACRO FACILITY	21-59
6.6.1 SMACRO Calls	21-60
6.6.2 SMACRO Definitions	21-60
6.6.2.1 Micros, Local & Global Symbols ...	21-61
6.6.2.2 Begin & End a SMACRO Definition ..	21-62
6.6.2.3 Value Specifications	21-63
6.6.2.4 Define Part Word Fields	21-64
6.6.2.6 Full Word Definitions	21-67
6.6.2.7 Conditional Control	21-67
6.6.2.8 Block Control	21-68
6.6.3 SMACRO Opcodes	21-68
6.7 COMPASS COMPATIBILITY.....	21-68
7.0 CRADLE REQUIREMENTS	21-69
7.1 FETS AND FILE BUFFERS.....	21-69
7.2 CONTROL STATEMENT OPTIONS AND FLAGS	21-69
7.3 UTILITY ROUTINES	21-72
7.3.1 Compiler Utility Routines	21-72
7.3.2 Host Supplied Utilities.....	21-72
7.3.3 SCOPE 2 Comdecks.....	21-72
7.4 DEBUGGING FACILITIES	21-73
7.4.1 Compile Time Reprieve Macros	21-74
8.0 OPERATIONAL ENVIRONMENT	21-76
8.1 MEMORY LAYOUT	21-76
8.2 WORKING STORAGE	21-77
8.2.1 F\$MEM	21-77
8.3 TABLE MANAGEMENT	21-77
8.4 CCG ENTRY POINTS OF INTEREST.....	21-79
8.4.1 Entry Points Refed From the Bridge Only	21-79
8.4.2 Entry Points Referencable From Bridge or End Processor .	21-80
9.0 COMPILER BUILD ENVIRONMENT.....	21-83
9.1 GENERAL BUILD PROCEDURE FOR A HOST COMPILER ...	21-83
9.2 SYSTEXTS	21-83
9.2.1 CMPLTXT	21-83
9.2.1.1 Macro Definitions	21-84
9.2.1.2 Symbol Definitions	21-84

9.2.1.3 Micro Definitions (from OPTIONS)	21-85
9.2.1.4 UPDATE Options	21-85
9.3 COMDECKS SUPPLIED BY THE HOST COMPILER.....	21-85
9.4 HCOEFS.....	21-85

[[] PAGE BREAK-- IV

1.0 INTRODUCTION

1.1 OVERVIEW

The CYBER 170 Common Code Generator (CCG) is a program consisting of a code transformer to do code optimization and a one pass assembler. It is capable of serving as the back end of algebraic language compilers such as PL/I, FORTRAN, C ...

The aim of this part of the IMS is to give the reader an explanation of the problems that CCG is attempting to solve and the methods it uses. A current listing is the present description of all details such as register assignments* etc.

The Interface Specification Appendix discusses the format and content of the interface between CCG and its host compilers.

In the environment of the host compiler CCG is responsible for the following tasks:

- Code Transformation - transformation of the CCG IL sequences into CYBER 17x machine instructions.
- Code Optimization - improving the efficiency of the object code by performing equivalence preserving transformations.
- Address definition for active transfer labels.
- Local block length computation.
- Relocatable binary output (assembly phase).

1.2 CCG-INPUT

This topic is discussed in detail in the Interface Specifications Appendix. Chapter 4 describes the Bridge input to CCG and 8.4.2 describes the available CCG routines.

[[] PAGE BREAK-- 1-1

2.0 ENVIRONMENT

CCG runs on a Cyber 70 or 170 computer & produces code for the appropriate Cyber computer.

The available resources consist of the X, B & A registers & small and large core memory.

2.1 RESOURCE SET (REGISTERS & INSTRUCTIONS)

In general the complexity of the problems increases as the number and variety (inhomogeneity) of the resources increase.

One of the reasons that a code generator is more complicated than the front end of a compiler is

that it has to interface with the machine and its resource set.

The code generator may be looked upon as a program that solves resource allocation, scheduling and packing problems. Likewise, an optimizer attempts to minimize the cost of a set of instructions by applying appropriate transformations.

The methods used to solve these problems in CCG are fairly general and we will now review some of the theory and terminology used.

Graphs and networks appear in various contexts in CCG. Subprograms (procedures) are represented as flows graphs in the global optimizer.

Sequences of instructions are treated as networks in the scheduler.

A graph is a set of points or nodes and a set of edges (lines) which connect the nodes.

A directed graph has an order associated with the edges* which may be represented as ordered pairs of nodes.

Mathematically a graph may be represented as a partial order on the set of nodes, i.e. a function $f: R \rightarrow R \times R^*$ where R is the set of nodes. Let x & y be nodes then x is the predecessor of y , and y is the successor of x .

Let $S(x)$ denote the successors of x and $S^{*-1}(x)$ the predecessors of x .

A graph is connected if R can be obtained by successive applications of S and S^{*-1} .

Throughout the IMS we will be interested in directed, connected graphs.

In global flow analysis we use control flow graphs to analyze the program structure.

Here the nodes are the basic blocks of the program and the edges are the control flow paths.

[[] PAGE BREAK-- 2-1

A path in graph is a set of edges (n_1, n_2, \dots, n_k) such that $n(e+1)$ belongs to $S(n(e))$.

A cycle or loop or closed path is a path at such that n_1 is in $S(n_k)$.

A graph without cycles is called acyclic.

The operand dependencies of an arithmetic expression may be represented as a Directed Acyclic Graph or DAG for short

A subgraph R is subset of nodes G and edges $E \supseteq 1$ of E such that $S(G)$ is a subset of G .

A strongly connected region of a directed graph is a directed subgraph in which there is a path from any node to any other node.

Program flow graphs have two distinguished nodes, the entry and exit nodes.

The entry node has no predecessors and the exit node no successors.

Program flow graphs are drawn in top to bottom orientation, with backward branches or paths on the left side and forward branches on the right side.

ENTRY

0	1	SUBROUTINE SUB
		REAL A(20)5
0	2.	I = 1

```

+>+>+ 3      10 IF(A(I).GE.0) GO TO 20
| | |
| 0 | 4.      A(I) = -A(I)
| | |
+>+>+ 5      I - I + 1
|            IF(I.LE.20I) GO TO 10
0 6          RETURN
EXIT          ENO

```

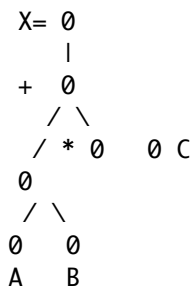
[[PAGE BREAK-- 2-2

For the above example

Edge Index Table Successor List

+-----+-----+-----+-----+			
I	Index to Succ List	Index	Pred# Succ#
	1	1	1 2
	2	2	2 3
	3	3	3 4
	5	4	3 5
	6	5	4 5
	8	6	5 3
	8	7	5 6
	8	8	6
+-----+-----+-----+-----+			

X = A » 8 ♦ C EXPRESSION TREE DIAGRAM



A network is a DAG with two distinguished nodes,
a begin node, which has no predecessors and a terminal node which has no successors.

While graphs may be represented by Boolean matrices where $A_{ij} = 1$ if there is an edge between the nodes i and j , this is inconvenient since the graphs we will deal with would result in sparse matrices. The most convenient format for our purposes is an edge index table pointing to a list of successors or predecessors. It is compact, easy to form and can be sorted easily, if it is not packed too densely.

Graphs are two dimensional in nature and in various instances we wish to impose a one dimensional or total order on the nodes in a graph. One example is the linear ordering of the instructions in an arithmetic expressions. Let the nodes of a graph represent instructions (operands or operators) and the edges

operand dependencies. An acceptable total order is one where all the predecessors of a node come before it.

The output of the sort is a list of the nodes in the order that they were placed on the output list.

We now consider two ways of obtaining such an order. The first is to represent the graph as a network and perform a topological sort. To do this we associate a predecessor count with each node.

[PAGE BREAK-- 2-3

Initially the begin node is placed on the output list and the pred count of its successors is decremented by one. Then all nodes whose count goes to zero are added to the output candidate list. We now select a node on the candidate list, move it to the output list, decrement the pred count of its successors, and add nodes with pred count equal zero to the candidate list and so on until the sort terminates.

During the sort a node may be in one of the following states:

1. Pred count is now zero.
2. Pred count is zero and it is not yet on the output list (on output candidate list).
3. Pred count is zero and it is on the output list.

Note that a topological sort is -

1. "breadth first" or "bottom up"
2. the order is not necessarily unique
3. time to sort is \geq the number edges
4. "maximizes the use of resources"

For a further discussion one may consult Knuth Vol I, pages 258-265.

Another way to order the nodes of a DAG is a "top down" or "depth first". In this algorithm we start at the top and visit the successors of a node until we find a node that has none. It is placed on the output list, the successor count adjusted and we backup to a previous successor, etc. The implementation of this algorithm requires a pushdown stack* for each node, a count of successors not yet visited, and the successor lists for each node. The algorithm is as follows:

```
i = 1      /* node number

1 IF( MSUCC(i)=0 ) GO TO 2
  MSUCC(i) = MSUCC(i) - 1
  STACK(i)  /* LET i BE A SUCCESSOR OF i THAT HAS NOT BEEN VISITED
GO TO 1
```

```

2  OUTPUT(i)
   POP(i)
   IF( i NE 0 ) GOTO 1
   END OF ALGORITHM

```

A depth first search is top down and produces a "narrow ordering" of the nodes.

For a further description* consult the Hecht-Ullman article or those by Tarjan.

[[] PAGE BREAK-- 2-4

2.2 PERT-TIHE AND INSTRUCTION SCHEDULING

PERT (Program Evaluation and Review) originated as a method for managing large projects, but it is also used in the compiler to schedule instruction sequences (BDT & MCG) as the 6600 & 7600 are capable of executing multiple instructions at a time.

A PERT network consists of a DAG with two dummy nodes. The nodes are called activities and the edges are dependency constraints.

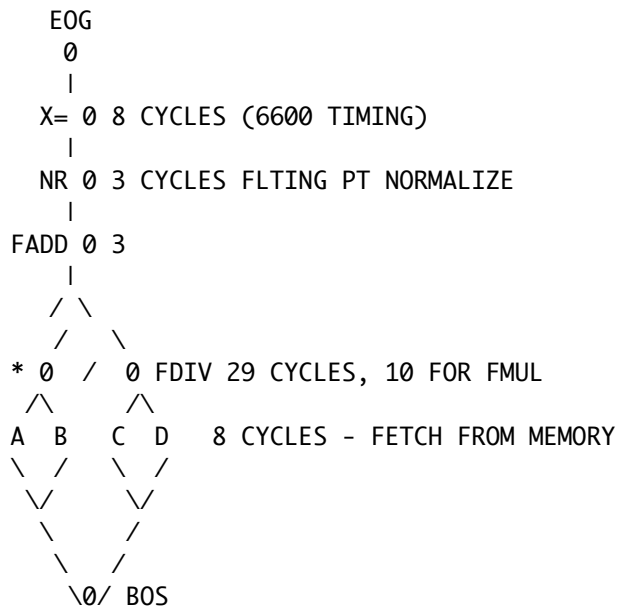
Associated with each activity is a time to perform the activity.

Associated with the network is a total time to complete it.

Consider the FORTRAN statement $X = A \gg B + C / D$.

The instructions are the activities, the execution time of the instructions are the activity times and the operand dependencies are the edges of the graph.

The questions which we are interested in are Given that a 66/7600 has independent functional units which may execute instructions in parallel* how may we reorder a sequence of instructions so as to minimize execution time.



The total time of all the instructions is $4+8+10+29+3+3+8 = 85$
 while the time of the longest or CRITICAL path is $8+29+3+3+8 = 51$.

We now make the following definitions.

The earliest start time (EST) of a node is the earliest possible time the activity can commence.
Mathematically

$EST(N) = \text{Max} (EST(p) + T(p))$ -for all p that precedes n
 $T(p)$ = time to complete activity p.

The latest start time of an activity is the last time that it can start without delaying the network.

[[] PAGE BREAK-- 2-5

Starting at "EQQ" compute the negative late start time:

$NLST(EQQ) = 0$
 $NLST(n) = \text{MAX} (NLST(s) + T(s))$ for all S, S succeeds n.
 $TOTALT = NLST(BOS)$ network time then
 $LST(n) \leq TOTALT - NLST(n)$ for all nodes in the network

Activities for which $EST(n) = LST(n)$ are on the critical path,
since any delay in starting the activity will extend the total network time*

[[] PAGE BREAK-- 2-6

3.0 GENERAL THEORY OF OPTIMIZATION

An optimizing transformation of a computation (expression, loop, program* etc.) is one that makes it possible to compute it more cheaply (faster) than the original, but produces the same results.

That is, we are interested in equivalence preserving transformations that minimize the cost of computing a result.

One can minimize the time or space cost of a computation. Our primary emphasis is on the former.

Over the years a fairly standard terminology has evolved for the various transformations and categories.

Machine independent optimizations are those that are profitable (reduce the cost of a computation) on any computer.

A simple example is the motion of invariant (loop independent) code out of a loop.

	$T = X \bullet Y$
DO 10 I = 1,N	DO 10 I = 1,N
10 A(I) = X • Y	10 A(I) = T

N-1 multiplies of $X \bullet Y$ are saved in the above right.

Machine or architecture dependent optimizations are profitable on some machines* but not others. For example instruction scheduling is profitable for 66 and 7600's but not for 6400's.

Local optimizations are transformations which are applied after analysis of a small part of a program* such as within a statement or basic block.

Examples are common subexpression elimination, compile time evaluation of constants, etc.

Global optimizations are those that are applied after analyzing a subprogram or some -large- part of it.
Examples are code motion, strength reduction, etc.

Most optimization centers around improving the efficiency of code in loops as the code in it is usually executed with a higher frequency than the code outside it.
Lacking further information from the programmer this is the fundamental assumption, and determines the strategy of all optimizing compilers.

Optimizing transformations can be divided into following general categories.

Code motion, which moves a computation from a high frequency region of execution to one with a lower frequency.

Strength reduction* in which a computation is replaced by a cheaper one.

[[PAGE BREAK-- 3-0

Redundant operation elimination, in which duplicate (common) operations are performed once.

Simplification and evaluation at compile time. (Calling sequences, constants, etc.).

Elimination of unused or useless operations.

Parallel computation of expressions (scheduling).

Register assignment.

Code motion usually refers to the motion of a loop independent computation from the loop body to the loop prologue.

Global register assignment is the motion of loads and stores of scalar variables in a loop to the entry/exit nodes of the loop and their assignment to machine registers in the loop.

Strength reduction is used in two contexts.

In general, it means replacing a computation with a cheaper one.

For example* $X**2$ is cheaper to do as $X*X$ than to call the library function.

Since addition is usually cheaper than multiplication* $X+X$ is cheaper to compute than $2.*X$.

The other usage of the term is the reduction of integer multiplies of a recursively (iteratively) defined integer variable in a loop and replacing them by adds.

EXAMPLE

DO 10 I = 1,10	DO 10 I = 1,10
10 N(I) = 25 * I	N(I) = IC
	10 IC = IC + 25

where I is iteratively defined and $25*I$ is a linear function of I.

Other types of loop optimizations are discussed in Allen's article in Rustin, & in Lowery & Medlock.

3.1 MEMORY REFERENCES, SEMANTIC INFORMATION AND DATA INTERFERENCE

The semantic information in a memory reference consists of the

IH - symbol table pointer
CA - constant addend or Bias Added to the base address
RF - subscript value (USUALLY AN R-NUMBER)

The source code $=A(F)$ produces the indexed load LD RI,RF,-1,A which is a reference to any one of the locations $A(1), A(2), \dots, A(N)$; where N is the dimension of A, and RF is the value of F. In the ensuing discussion we will assume that different names refer to distinct (non-intersecting) classes of memory Locations.

[[] PAGE BREAK-- 3-2

During optimization one may wish to move or reorder instructions. In order to move memory references without changing the meaning of the computation we must have rules for doing so.

Two data references interfere with each other if:

- a. at Least one is a definition
- b. the intersection of the classes of memory locations referenced by them is not the empty class.

Consider the following examples* where the values of I and J are unknown, so we must make the worst case assumption that $I = J$ is possible:

1. $X = A(I)$ $A(I), A(J)$ interference
 $A(J) = B$
2. $A(I+1) = B$ no interference
 $A(I) = C$
3. $A = B$ interference
 $X = A(I)$
4. $X = A(I) * A(J)$ no interference

Restated in terms of IH, CA and Rf's two data items interfere if:

1. At least one is a store
2. IH's are the same (same SYM table entry)
3. Rf's are not the same or Rf's & CA's are equal

Minimizing the data interference permits maximum optimization to take place without

having to collect and analyze extra information about the range of subscript values. As an example* consider the loop:

```
DO 10 I = 1,N
10 X(J) = X(J) ♦ A(I) * X(J+I)
```

which is not fully optimized by the compiler.

By changing the references to X(J) to a temporary variable inside the loop

we get rid of the data interference that the compiler sees and improve the code.

```
T = 0
DO 10 I = 1,N
10 T = T + A(I) * X(J+I)
X(J) = X(J) + T
```

[[] PAGE BREAK-- 3-3

Data interference considerations arise during memory reference squeezing, code motion, global register assignment, scheduling, and use-definition analysis.

In each case the specific definition of interference is slightly different.

The above definition is used for common subexpression squeezing and scheduling.

Note that because of the simple structure of references in FORTRAN the computation of data interference is fairly simple. In PL/I the problem is complicated by the existence of based arrays.

3.2 REGISTER ASSIGNMENT AND DEADLOCK SITUATIONS

Some optimizations* such as common subexpression squeezing* may give rise to deadlock situations that may become apparent in some later optimization process, usually register assignment. The classic situation that may arise is that of over squeezing, followed by register assignment. During the register assignment phase it is found that two quantities are attempting to occupy the same register at the same time.

As an example* consider the sequence (decimal #'s, R-nums are multiples of 4)

DEF 4,B1	B1 holds a result
DEF 8,B2	B2 holds some other result
STT 12,4,1,A	short add of B1 + 1 +A
RS 12,82	put result into B2
STT 20,8,2,A	uses B2 which previous STT destroyed
RS 20,B1	

Which once came about from merging the epilogue/prologue of two similar loops & "over squeezing" of STT's in SQZ.

The basic principal is the uses of one variable overlapping with the definition of another which is to occupy the same register.

Most of the logic in CCG is setup to avoid these situations by constraining SQZ, etc. so as to avoid potential deadlock situations.

By doing things in this fashion CCG misses some chances to produce optimal code.

As another example* consider the sequence

```
I = I + 1
A(K) = SHIFT (W,I)
X = A(I)
```

[[] PAGE BREAK-- 3-4

One would like to rewrite the sequence as

```
IOLD = I
```



```

I = I + 1
A(K) = SHIFT (W,I)
X = A(IOLD + 1)

```

Which would shorten the critical path* if we could get rid of IOLD. But by getting rid of it we would have one variable I, whose different values I and I+1 are needed after its redefinition. Note that the load of A(I) cannot be moved up because of the interfering store into A(K).

[[] PAGE BREAK-- 3-5

●●●SIJ NOTE May 1,2024 - Many of the below are unavailable or expensive.
A recent copy of Compilers ... by Aho, Sethi & Ullman should be a decent reference work.

3.3 REFERENCES (N.B. - MOST ARTICLES & BOOKS PUBLISHED BEFORE 1980)

The following books and articles were found to be useful.
They may be consulted for algorithms, general background, terminology, etc.

Rustin "Design and Optimization of Compilers-, Prentice Hall
Excellent article by Fran Allen* -A Catalog of Optimizing Transformations-

Gries "Compiler Construction for Digital Computers", Wiley.
Good Text* Chapters 11-13 and 17-21 are of interest.

Cocke and Schwartz "Programming Languages and Their Compilers" NYU (CIMS) Notes.
Chapter 6 is of interest* beware of typos & logic errors. Contains a description CSC LNRA, interval analysis* etc. (

Aho-Ullman -Theory of Parsing... Vol II" Prentice Hall.
Very abstract, good description of basic interval algorithm and good bibliography.

Knuth "Art of Computer Programming* Vol I" Addison Wesley. Good reference.

SIGPLAN July 70, Vol 5, #7 "Symposium on Compiler Optimization"
Good articles by Allen, Bagwell, Frailey.

1 SIGACT/SIGPLAN, Oct. 73 - article by Hecht & Ullman on
the use of bit vectors for global flow analysis. Good bibliography.

R. E. Tarjan - article on Depth first searches in SIGACT May 1973.

F. Allen* "Program Optimization". Annual review in automatic programming, Vol 5, Pergamon Press, 69.
Classic article on optimization.

Lowery & Medlock, "Object Code Optimization". CACM, Vol 12, #1, Jan. 69
Good reading* but not enough detail to be really useful●

J. C. Beatty, "Register Assignment Algorithm..." Jan. 74 IBM Journal of Research & Development.
Hairy, but worth trying to read.

C. P. Earnest* "Some Topics in Code Optimization", JACM, Jan. 74

Conway et al, -"Theory of Scheduling" Addison-Wesley. This is the classic text.

1 Beatty, "An Axiomatic Approach to Code Optimization" JACM Vol 19, #4 (Oct 72).
C Minimum depth parses and such.

[[PAGE BREAK-- 3-6

4.0 CCG IMPLEMENTATION LANGUAGE

The majority of the code in CCG is written in assembly language defined by the COMPASS assembler. The only exceptions are the test mode routines that print formatted structure and table dumps which are coded in FORTRAN. The main reasons why COMPASS was chosen as the implementation language, size and execution speed.

There was no high level language available that does not seriously impact both the size and speed of the resulting product. There are, however, several disadvantages to coding in COMPASS: Increased development time, maintenance expense* and loss of readability are the most serious. Also the standard form of commenting the action of a machine instruction in English is imprecise. In an effort to minimize these disadvantages a low level algorithmic language* similar in notation to PL/1 was developed for use in designing, coding, and commenting CCG.

This CCG Implementation Language is described in the following section.

When it is used properly and faithfully throughout the code it gives visibility to the algorithms, the data structures being accessed* and the general program flow.

It is vital to maintaining the integrity of the code that all changes be made to the algorithmic language notation as well as to the COMPASS instructions.

For quick reference to the algorithmic comments a utility program* FTNDOCK has been developed to produce concise

formatted listings of the routines as they appear coded only in the CCG Implementation Language.

●●●SIJ NOTE Informally the Language was called JAW which could stand for
Joseph Allen White (my mentor on the FTN project) or "Jasik and White"

To properly use the CCG Implementation Language it is important to understand that it is not Just a means of writing shorthand comments after the COMPASS code for a routine has been completed.

I used the language to sketch the initial design of a routine, the problems of register assignments, instructions,

code sequences, etc. are avoided until later.

Once the routine has been completely coded in the CCG Implementation Language and thoroughly desk-checked it is a fast, efficient procedure to translate the algorithmic language into assembly language instructions.

By preserving the original CCG Implementation Language instructions in the comments field of the resulting COMPASS,

the algorithm used, data structures accessed, and general flow of the routine are readily apparent to the reader.

By associating a variable name with the contents of a register its uses and lifetime are more obvious. The final combination of assembly language instructions and algorithmic language comments provide the readability of a higher level language while preserving the size and speed of the compiler.

The following rules describe the way the CCG Implementation Language has been used in most of the routines. Since the language was extended as new uses were found it is not entirely consistent throughout all of the code.

However, an effort has been made to give a complete and concise description of the language here.

[[PAGE BREAK-- 4-1

GENERAL NOTATION AND FORMAT

●●●SIJ NOTE In 1972 - 1974 text editors didn't exist so I used the IBM 029 keypunch machines in the room where the women produced the 80 column cards to be fed into the computer via the CDC 405 card reader.

I was quit proficient at that as the compiler consisted of around 20 boxes of 2000 cards per.

The code should be punched with an 11-18-30-36-44 drum card. The algorithmic statements begin in column 30 or 36. Their comments field begins in or after column 44. To identify a comment the first two characters must be "*/" & they are followed by a blank. Variable names should be kept short (2 to 4 characters).

All tables and data structures should be referenced by the same mnemonic used in the COMPASS code to avoid confusion.

All routines are also referenced by the same name.

The basic style of the statements is similar to PL/1.

Arrays may be base 0, while most fixed tables are base 1.

Many of the variables represent address values. Consequently the following notation was adopted to indicate indirect addressing: If X is a variable then [X] means the contents of X.

This notation may be used to indicate a load of a value as in: DI = R1+2, DIW = [DI] where DI is the address that is the address R1 + 2 and DIW is the name given to the contents of DI.

The same notation is used to indicate a store: [DI] = RDT(OC.XMT) v 1

means that the value of the computation on the right hand side is stored at location DI.

REFERENCES TO BITS AND FIELDS OF DATA STRUCTURES

Bits and fields of data structures are defined by using the DESCRIBE, DEFINE, and DEQU macros defined in the comdeck COMADEF.

For example consider the definition of the structure Z. Which occupies one 60 bit word:

	DESCRIBE	Z.,60 ZW(TB,TF,MF,3F)
TB	DEFINE 1	Top bit
	0EFINE 5	unused - 5 bits
TF	DEFINE 18	Top field
MF	DEFINE 18	middle field
BF	DEFINE 18	bottom field

This defines the Z. structure to have four fields* TB, starting at Bit 59, TF starting at Bit 36, which is 18 bits long, etc. Since TB is 1 bit long it is thought of as a logical variable.

Now consider the following comments.

X = A+2	address computation
TB[X] = 1	field assignment
IF(TB[A]) GO TO 10	field reference - IF TB[A] = TRUE
IF (BF[A] NE 2) GO TO 10	field reference
[X] = ZW(1,2,3,4)	field combination & assignment

So we see that if F is a field name* then FIX) denotes extraction/ assignment of it.

If ZW is the structure name* then =ZW(A,B,...Z) denotes combination of the fields A through Z.

[[] PAGE BREAK-- 4-2

The naming restrictions for fields is that the number of chars be less than 3 or 4,
and that the structure prefix be 1 or 2 characters.

When a field F in a structure X is referenced the word from the field is given the structure name,

so for example a reference to the BM field in the descriptor word of an IL instruction would be BM[DI], where D is the structure prefix.

Most of the data types used in the compiler are integer, with the exception of the one bit fields which are logical variables.

The operations are the basic arithmetic and logical operations and simple & multiple assignment. Multiple assignment to two fields in the same word are indicated by -

(A,B)[W] - (VA,VB)
OR
(A,B) = (FA,FB)[W]

References to the IL pervade CCG. The four IL words are:

R1 R-one word
R2,IH - R two or IH word
DI,DJ - descriptor word
LI,LJ - link word

The important fields of an IL instruction are the R-numbers or operand links: RI, RJ, RK, R, RN ...

Loop counters and limits are usually given single letter names I, J, K, L, M, N.

Temporary variables are used to give a short name to a table element, field, etc. that will be used in a later reference or definition.

A common suffix for many variable names is W for word.

For example

BIW = BIT(I) is the block index word.

As can be seen from the above discussion* the notation is and can be ambiguous, but an understanding of the structures being manipulated and an inspection of the comments should make them clear.

REFEFRENCFS TQ TABLE ELEMENTS

Table elements may be referenced with two different notations, as A(I) or [A+I].

For example* the first notation is similar to FORTRAN array references.

TI = TREE(I) where TREE is a table and TI holds the current value of the element at index I.

Suppose a loop to scan the TABLE would be written as follows in FORTRAN :

```
I = 1
10 I = I + 1
   IF(TREE(I) .NE 0) GO TO 10
```

[[] PAGE BREAK-- 4-3

The same loop is more frequently written in the CCG .Implementation Language as:

```
TI = TREE          */ Set up address register
LP TI = TI + 1      */ Advance to next element
IF( [TI] NE 0) GO TO LP  */ Test for end of table
```

This case illustrates how the index variable I can be subsumed into the address reference

of TREE(I) .

This type of notation is more compatible with the assembly instructions.

CALLS TO SUBROUTINES

Subroutine calls and function references are noted in the following manner:

```
CALL SUB(A,B,C) */ Call subroutine SUB
Z = F(X)        */ Reference function F
```

where A,B,C are the variables being passed to or returned from the subroutine call and X is the argument to the function reference.

Basic FTN intrinsic functions are called by their standard names.

(SHIFT, MASK, NORMC, MAX, MIN, etc.)

CONDITIONAL BRANCHES AND ASSIGNMENTS

IF statements are used to show conditional branching. They are written in the form IF "condition" .

IF statements are also used to show a conditional test being used to determine a value.

In this case an assignment may be used in the algorithmic language:

```
♦      LT 82,B3,+1 J = MAX(J,K)
```

or

```
A = IF(LOGEXPR) THEN TrueV ELSE FalseV which is the ALGOL conditional assignment.
```

[PAGE BREAK-- 4-4

4.1 DEBUGGING FACILITIES (SIJ Note: DECK is CDC speak for a file covering a "topic", core = Memory)

CCG contains extensive debugging facilities in test mode.

In addition to the normal register snap and core dump facilities available throughout the compiler, CCG has a set of formatted table dumps. These debugging printouts are activated by inserting macro calls and reassembling one or more decks (FILES).

In test mode there is the added feature that termination dumps of various compiler tables and structures are printed whenever a compilation is aborted. These are useful in determining what type of processing was being done when the error occurred.

The SNAP, SNAPT, REG, and PRINT macros are described in CCGTEXT .

There are also several other special PRNTx macros that are defined in the CCG decks that use them (such as PRNTM in MIO). These macros offer a convenient method of calling one of the FORTRAN coded debugging routines to print out a compiler table or structure in an easily interpreted format complete with mnemonic headings. The FORTRAN routines that produce the formatted output are grouped together under the heading CCG Utilities in the following section on Deck and Routine Descriptions. Most of the PRNTxxx routines are associated with a particular deck, while the OMPxxx routines are associated with a structure.

The TRACE macro is a convenient way of leaving table snaps in the compiler. The macro format is

```
TRACE LAB,TBL,BLK
```

where LAB is the trace label, TBL is the name of the table to be printed in octal format unless TBL = RLIST in which case the table is printed as IL instructions which are assumed to be in

table TXT unless the BLK parameter is present and specifies another table.

Standard sets of debugging snaps have been permanently installed in all major routines. These have been selected to trace compiler flow, snap IL sequences in various stages of modification, and dump pertinent tables during execution.

The TRACER macro is used to specify a list of debugging printouts to be activated within a routine. For example:

TRACER (A,B,C)

activates the printouts specified by the TRACE labels A, B, and C.

A complete list of debugging labels in a routine may be found by checking the symbols in the qual block DEBUG in the reference map. To build a test mode compiler with all debugging routines assembled in and a DEFINE TESTCCG directive should be added to the OLOPL, and the compiler and the texts rebuilt.

In the following routines a subset of these debugging snaps may be activated by reassembling the routines with an UPDATE "*DEFINE deckname" directive: MIO, F3V, GPO, GRA, SQZ, MCG, B0T, CFA, & UDT.

This will conditionally update in a TRACER directive specifying the most frequently used snap labels. Since snaps and formatted dumps can produce a large amount of output it is recommended that test cases be reduced as much as possible before activating snap labels.

[[] PAGE BREAK-- 4-5

4.2 INTERNAL LANGUAGE DESCRIPTION

The IL is described in sections 4.3 to 4.3.6 of the Interface Specifications Appendix with the exception that certain instructions that the host should not use are not discussed there.

The following completes the information in those sections.

4.2.1 OPRDEFS

The comdeck OPRDEFS defines the name, order and properties of the IL instructions.

In CCGTEXT and CMPLTXT the comdeck CGGILFD calls it to define the opcode values (OC. symbols).

The IL instructions are divided into the following groups -

1. Pseudo instructions EQQ - ENT
2. Machine instructions XMT - CX
3. Memory references & sets LD - DWL
4. Conditional Jumps: JPX, JPBB
5. Unconditional Jumps: JIN - UJP
6. Optimizer instructions: ILD - SXT

The order of the instructions in OPRDEFS is critical. Any change must NOT violate the following constraints:

OC.EQQ = 0 assumed everywhere

OC.BOS = i assumed everywhere

DAR is the only type I instruction less than OC.RS (BDT/CRW)

For group 2, the machine opcode = IL opcode.

Memory reference instructions always occur in the order load, store, set

LD is the first memory ref. DWL is the last.

JIN is the first unconditional Jump, (BOT/FTL) & conditional Jumps come before JIN (MCG/PJI)

Add, subtract instructions are next to each other. Assumed in SQZ.

Instructions >= OC.CLR do not show up as input to CII (CGIM/CII)

1 Current definition of the layout of the fields for the various instruction types is in CCGTEXT.

[[PAGE BREAK-- 4-6

4.2.2 CCG ONLY INSTRUCTIONS

DAR Define A-register pseudo instruction.

Used to associate an R-number with an A-register of a previous load instruction.

This pseudo op will be used for prefetching/Loop folding only (GRA/CLB, MCG/PPI).

SLO. RI,RJ,RK SAI AJ/BJ+BK SHORT LOAD

SST. RI,RJ,RK SAI AJ/BJ+BK SHORT STORE

SOL. RI,RJ,RK SAI AJ/BJ-BK SHORT DIFFERENCE LOAD

SDS. RI,RJ,RK SAI AJ/BJ-BK SHORT DIFFERENCE STORE

DRL. RI,RJ,RF,CA,IH RXI XJ OIRECT READ LCM | 7600 |

DWL. RI,RJ,RF,CA,IH WXI XJ OIRECT WRITE LCH | ONLY |

ILD initial load of a quantity, Synonymous with a *LD»,
except that in "SQZ" a ST/ILO combination kills the store after squeezing the IL.
RF must be zero

TLD/TST. Temp LD/ST instructions introduced by GP0. Always of the form TLD/TST RI,0,CA,ORD(IT.)
CA is an index to »TET».

4.2.3 THE S0 FIELD & REGISTER SPECIFICATIONS

The S0 (Specification Ordinal) field exists in all type II instructions,
but is meaningful only in the »RS» and »DEF» pseudo instructions.

The format of the S0 field is -

4/unused,2/inv_bits,2/lock type, 3/reg_type,3/reg_num

The register type values are 0,1,2 for B, A & X registers. They may not be changed.

The invariant bits are set by GP0/GRA

and are used in GRA (ERC) to extend inner loop register assignments to an outer loop.

The types of register locks are -

0 -UJP lock (prior to a function call RJX/JIN/UJP)

This type of lock is used to ensure that the arguments are that are passed in registers, stay there during code scheduling until the RJX/ JIN/UJP is issued.

1 -temp lock. Place the result in the specified register, and keep it there until its uses count goes to zero.

Operations that precede a temp lock RS may be eliminated in SQZ if they have no uses.

[[] PAGE BREAK-- 4-7

2 -full lock. This lock type is used to hold a result in a register until it is overridden by a second full lock RS (and uses =0), or to end of sequence.

Full lock RS's and DEF's are used by the global optimizers (GRA and AIS).

3 RJ RS lock. This type of lock and its RS and are used following UP, NR instructions that produce 2 results (X and B).

The RS is used to specify the B-register result and hold the uses count. For a RJRS the IN field in the R1 word is 1.

3 A-lock* used for full lock and definitions on A-registers that are setup by GRA (GRA/IRA and MCG/PII).

4.2.4 REGISTER REDEFINITION

When two or more RS's to the same register occur in a sequence

it is necessary for "BDT" to form logical links in the dependency network between the instructions using the result of the first RS and the operation defining the second. When both RS's occur in the same basic block BDT does this automatically.

When the second RS is in a different basic block, then the first & redefinition links are necessary, as in the case where a loop index is kept in a B-register* one may place the value of the first R-number in the RF field of the IH word of the second RS.

EXAMPLE:

STT 4,,,A

RS 4,,B7

LAB ,LOOP

SLD 20,4,, (IH(4,0,A)

FM 24,20,RK ---

STT 30,4,1 B7 = B7 ♦ 1

RS 30,,B7 IH(4*0*0I

Other fields that may be set in the initial RS,DEF instructions are the IH,CA fields in R2 words to specify the variable in the register.

In redefinition RS's and initial DEF's the CA field is used to hold the part of a constant that is in a B-register. The situation arises when an address is placed in a B-register and the sequence contains PLD/PST's.

Consider a loop with references to A(I) and A(I+5) where A+I-1 assigned to a B-register:

	OEF	4,B7	CA=-1
A(I)	SLO	10,4,0	IHW (4,-1,A)
A(I+5)	PLD	14,4,0	IHW (4,4,A)

The CA in the IH word is the semantic CA, the real CA is $4 - (-1) = 5$

[[PAGE BREAK-- 4-8

5.0 TABLE AND STRUCTURE DEFINITIONS

The general table layout for CCG is discussed in Chapter 8 of the Interface Specifications. What follows here is a description of each table used by CCG.

"Fixed" FWA tables in FWA order (OPT=2 only).

HNT Header Node Table. For each loop it holds the bit index of the header and the HB (holding block). It is used by FXI to find the HB number when an exit node from one loop jumps to the header node of another loop.

LCT Label Change Table. Setup by GPO/IRP, used by CII when an inner loop exits to the header node of an outer loop to change to label references.

IST/GST Interval (flow graph) structure tables. Formed in CFA, and used by GPO. It may reside on disk if large.

"Fixed" FWA tables in FWA Order (OPT = 2 ONLY)

BIT Block Index Table - 2 words per entry
For each basic block in the program. BI. and RI. word formats.
BI. word contains block properties.
RI. word contains address of block on mass storage or in core.
Initially the table is built in *BST* and the info filled in by *P83*.
Moved to low core and renamed by GPO.

UDT Use-def info table
final version* 2 words/entry for each referenced scalar variable or class of memory locations
First word is in UD. format, second is 12/packed shift count, 30/0, 18/word index,
that is used to access the entry for the variable in the bit vectors.
Formed in *UDT*f moved to low core in GPO. Referenced in GPO and GRA●

MVL Marked Variable List. List of pointers to UDT of the variables that are referenced in a block.
Formed in GPO/FUD, referenced in CRB. BVT - Bit Vector Table.
Four bit vectors per block for each block & holding block in the program.
Bit vectors are in the order OEF* UBO* USE* LX (live exit).
A bit vector is a string of bits occupying contiguous words.
The UD descriptor word is used to address bits in bit vector. Formed in FBV, GPO, & AUT.
Referenced/ modified in FBV * GPO * GRA. Note that each bit vector is *VL* words long.

SBV Special bit vectors that are used by GPO/GRA.

[[PAGE BREAK-- 5-1

BTT Block transition Table - Used to change a node number into a block number.
 Formed in GPO/IGP, referenced in GPO/FXI.

[[] PAGE BREAK-- 5-2

-DYNAMIC- TABLES (managed by table manager in CGTM)

BLK Dynamic table used to hold core copies of blocks.
 0.SEQ, L.SEQ point to the block that is currently being processed.

TXT IL instruction sequence.

GST name of IST when being formed by CFA

RND R-number definition table in SQZ.
 Scalar load address table in PRE (IXFN processor)
 Scratch in various subroutines
 in GPO (MIP,DIF) and GRA (MTA,CRN,OXA,MFA,CLB)

TREE Successor index table in BOT/MCG

PIT Posted instructions in MCG and CGTM/UIII
 SI. format in MCG/SII

OTI Optimizing temporary info in MCG/JAM

MLT Mod list index table outside of MCG.
 (1 word per modification in ML format (CMPLTXT))

MOO Table of modifications (IL instructions)
 4 words per instruction. First entry is a ♦BOS*. Used in conjunction with MLT♦

CFT Control Flow Table* 1 word per entry in CF. format.
 Contains the edges of the program flow graph (FROM,T0)♦
 Formed in the bridge. used in CFA. Also used by GPO as a scratch table.

BST Block status table 2 words per entry for each block in the region being processed.
 First entry is the holding block (if any), and last word is 0, a table terminator.
 BST is used as a loop control vector to hold the list of blocks that are being processed
 by the optimizers (GPO, GRA), and to pass information between the subroutines.
 The first word is in BA. format and points to the block.

PSI Post store info, a collection of lists of variables and the registers they are in,
 that are to be stored on entry to a block. Formed in GRA/SXC and referenced in GPO/IPS.
 The post store lists are in PS. format and are pointed to be the PII field in BIT.

RCT Register candidate table. Setup & used by GRA to pick & assign candidates to registers in loops.

RXI Region exit information. Setup in GPO/FXI for use by GRA (SEE: IRA & SXC).

TET Temporary equivalence table, 1 word per entry in T. format.
 TET(C) holds information about the status of the global temps (TLD,TST) with CA=C.

Referenced in UII, SQZ, GPO, GRA.

[[PAGE BREAK-- 5-3

IOL I/O lists, 1 word per entry accumulated for a basic block in the bridge
Appended to the end of the block by PBB.

IIT Increment Information Table, consists of three lists, see GPOCOM for a description.
Formed in GPO, GRA.

UDI Use Def Info, 1 word/entry in UA. format. Initial form of UDT.
Formed in UDT, reformatted in AUT.

CVT Constant Value Table*, 1 word/entry.
Contains binary value of converted constants that are referenced by LDC instructions.

CUT Constant Use Table* 1 word/entry, parallels CVT.
Entry is non-zero if corresponding entry in CUT has been referenced
in an issued instruction (CGTM/CII) or an Aplist (Bridge), etc.

Host Supplied -Static- Tables in High Core

The following tables may be static or dynamic as the host chooses.

GLT GL (Generated Label) address definition table.
CG\$LABD saves the block number in the table during phase 1 of OPT=2.
Normally it is used to hold the address definition of the label in WC. format.

CBT Common Block Table* 1 words/entry. First entry is blank common.
Format is described in 2.2.1 of the Interface Specifications Appendix.

SYM Host Compiler Symbol Table, 3 words/entry.
Described in section 2.1 of the Interface Specification Appendix.

5.1 STRUCTURES

Various structures appear in most of the routines in PASS 2.
This section is an attempt at listing all places where they are referenced.

BIT - Block Index Table

BIT is an in memory table that is a repository of information about the blocks that
must be available on a random access basis.
There is one entry for each basic block in the program, an entry for each HB,
and entries for the program exit block (block No. 0) and the pseudo entry (block #1).
Each entry consists of two words. The second word is the address of the block (disk, LCM, SCM),

[[PAGE BREAK-- 5-4

its length and where it is. This word is for MIO's use and is rarely referenced by the optimizers.
It has the property that the LEN field is always correct and may be referenced.
The first word is the block information word (BI. in CCGTEXT),

and it holds the block type (reachable program block or HB), format (C8=1, then in SI. format), final parcel count when coded, index to bit vectors in BVT, etc.
This word is referenced by most of the optimizers. One should note that the order of program blocks in BIT is their source program order. The IH field of the BOS at the beginning of a block is the BIT index of it. The block index of BI is twice the block number (BN) of a block.
In order to avoid confusion* one should know that the control flow analysis code usually uses BN's, while most other code references BI's.

During the bridge processing BIT is setup in BST by UDT/PBB.
After we have formed the interval lists and know how many loops there are it becomes a fixed table in low memory with pointers 0.BIT and L.BIT.

During the optimization phases BIT is referenced in FBV, MIO, GPO & GRA.

BIT VECTORS (USE/DEF)

Bit vectors or bit strips are used in the compiler for structures that may get very large.
The USE/OEF bit vectors are one example. For each block in the program there are four bit vectors that must be maintained during global optimization.
They are the DEF, UBD (used before definition), USE and LX (Live on eXit) bit vectors.
The first three are used to calculate the LX bit vectors.
The LX bit vectors are used to eliminate dead definitions, determine when variables must be initialized or stored on exit from a loop, etc.
Furthermore the bit vectors could also be used for global common subexpression squeezing* etc.

Bit J in word K (K=0,1,2,...) of a bit vector corresponds to the $60 \cdot K + J + 2$ entry in UDT.
The second word of UDT holds the bit vector -address- in the format: 12/P (bit number), 3Q/0.18/word index .

Most of the bit vectors are kept in BVT, which is setup as a fixed table in GPO initialization in SCM.
Because of their relatively small individual size and random referencing, they are kept in SCM.
A more flexible scheme would allow them to be kept in ECS/LCM.

Summary of references:

UDT/AUT Setup UDT, form special bit vectors in RXI (common variable -spoil vector-,
 dead def complement vector and DEF, UBD, USE, LX vectors for the program exit node).

GPO init allocate for BVT, move special BV's from RXI to BVT.
 Call FBV to form the bit vectors for the program blocks.

[[] PAGE BREAK-- 5-5

FBV call FU0 to form the DEF,UBD & USE BVs for all program blocks, then compute the LX BVs.

IGP initialize the BV address of the holding blocks in the loops
 to the loop entry block in case the exit node of a loop is a HB
 which has not yet been processed (needed in FXI).

IRP clear the HB BV address field.

TRP call FU0 to form the HB, DEF, UBD, USE BV's. HB live exit = region live entry BV.
 Save region BV's (DEF, LU, USE, LX) in header node slot for use by FUD, Mil, etc.

When processing an outer loop.

FUO form block DEF, UBD, DEF BV's if BBV = 0 and clear LX BV.
If block is a HB, then find the header node, and adjust
the region USE, DEF and LU vectors of this loop.

CHB combine HB with natural pred, adjust bit vectors of both blocks,
call AUV to adjust USE,UBD vectors to account for any ST/LD squeezing.

RDD scratch vector, SV= LX or (DBU and USE), dead defs = not SV and OEF.

IPS insert post stores, set DEF bit, clear UBD.

AUV rescan block and recompute USE, UBD vectors if no user extrefts in the block.
Makes the USE vector smaller if any ST/LD squeezing took place.

FXI form entry/exit info for the loop being processed.
Live entry = UBD or (not DEF & LX) in header,
compute live entry at successors of the loop.
Compute region live exit = OR LX over all successors.
Movable defs = DEF in region & not live entry, over successors that can't be post stored into.

MII/ST clear bits in block DEF vector, and set RF in ST so RDD doesn't KILL it.

MII/EOQ accumulate region DEF & USE bit vectors.

GRA/IPP Set IST for variables used in an inner loop (LU).

SEE Reference L entry, LX, movable def info and places it in RCT.

ERC References UDT, add variable to LUV if not entered as a candidate.

SXC Reference RXI to form post store lists adjust LU, LE BVs.

IRA/EOQ output epilogue post stores for variables that are LX from the epilogue, set bits in LUV .

[[] PAGE BREAK-- 5-6

TfT * Global Temporary Equivalence Table.

During invariant code motion and strength reduction GPO forms global temporaries.
During execution time they are kept in the IT. block and during compile time they are
referenced via TLD, TST and ILD instructions.
Since they are subject to elimination, squeezing and packing,
the CA of the TLD/TST instruction is an index into TET, the temporary/equivalence table.

N.GT is the number of global temps at the beginning of a loop, base of temps created in this loop.

Summary of references

GPO

GP026 - level of optimization, assign CA's to the non-equivalent entries in TET (packing algorithm).

MIE - form IT.(K) = EXPR in HB, save TST type & pointer to TLD insert point in TET.

CIF - Point from TET to IP formula info in IIT for strength reduction temps.

EIE - form variable increment temps and add to HB

ATT - clear REG field for TET's created in this loop for GRA, update info of equivalenced entries.

UPB - use TET as a control vector to insert TLD's of the removed expressions in the loop body.

SHB - call SQZB to squeeze the HB and the TST's.

SQZ/STS - squeeze temporary stores to eliminate those that are the same type & pointing to the same expression (multi-block and outer Loops), and assign temp numbers to the TET entries.

GRA

MTA - set REG field of TET's that were assigned to B-register for MFA.

MFA - Reference REG field when address differencing.
Reference TET when setting final test replacement decision.

SUP - Reference TET.IIT if TR0=4.
Adjust TET by clearing HBN field of TET's created in this loop. Scan HB and set HBN number of it's defined in this loop. Finally scan TET backwards and remove trailing entries with a zero HBN field.

OIL - issue ILD of a TET entry if created in current loop.

CGTM/WII - assign storage locations to TET's when first encountered (OPT<1 only)

[[] PAGE BREAK-- 5-7

SIJ NOTE - After 46 years this diagram doesn't make much sense to me, for completeness it is here

Memory Layout During CCG Processing
OPT=2

	Phase 1 and 2	Phase 3
RA+0	-----	
	code (CRAOLE, ENOPROC, ASSEMBLER, Code Transformer)	
		-- ist
	BRIDGE	BIT
	CFA	fixed UDT
	UDT	tables MVL
	-----	BVT & SBV
	UDB Buffer (100B)	BIT
	OPT=2 Buffer (2001B)	+-- LCT

	Dynamic Tables	+-- BLK
	-----	Dynamic TXT
	Static Tables	Tables RND
RA+FL	-----	

===COMMON BLOCKS===

CCGSCR - 260B words are scratch for use in a routine.

Rest is used by GPO and GRA. First 100B words used by SQZ. BOT, MCG & PROSEQ.

GPOGRA - used for communication between GPO and GRA.

5.2 GENERAL PROGRAM FLOW

OVERALL FLOW OF CONTFOL

See section 1.1 through 1.3 of the Interface Specifications Appendix.

PRIMARY FUNCTION BY DECK (FILE)

The Bridge controls the accumulation of IL instructions & calls CG\$PAS to process the accumulated sequences.

CGIA is the internal assembler.

CGTM contains the table manager, utility routines and the host callable entry points that are used during ENO processing.

PROSEQ controls the processing of the accumulated sequences & the expansion of memory references. It calls SQZ to remove redundant instructions & collect uses counts, etc.

[[] PAGE BREAK-- 5-8

In OPTs 0,1 PROSEQ then calls ESR to expand references to formal parameters and level 2 variables, MCG to code the sequence and WII in CGTM to write the machine instructions (in SI. format) to the SLIST file.

Finally, it resets various flags for the next sequence and returns.

In OPT«2 PROSEQ calls PBB in UDT to process the basic block.

PBB calls subroutines to chain the memory references to the UDI, the use def information table to form a block table entry for the block; and to save it on secondary storage (LCM or disk).

SQZ does common subexpression reduction, compile time constant reduction, instruction simplification & dead instruction elimination on an extended block basis.

MCG is responsible for instruction scheduling & local register assignment. It calls BDT for form the dependency graph, or to reorder the instructions.

GPO is the global optimization controller.

It contains the machine independent optimizers for invariant code motion, strength reduction & dead definition elimination.

GPO calls AUT to adjust the use-def table, CFA to analyze & reformat the control flow information, & FBV to form the use, def and live exit bit vectors.

MIO is the block manager. It is responsible for paging blocks in & out of core as necessary during global optimization.

FBV is called by GPO to form the live exit bit vectors.

GRA is called by GPO to do -global- register assignment for a loop after the machine independent optimization has been performed. It builds a table of candidates for register assignment, makes assignments based on frequency, usage and availability, and attempts to code the loop with the assignments until it is successful.

CFA analyzes the control flow information and forms the interval lists that direct the optimization in GPO.

UDT contains miscellaneous subroutines that are used during the first phase of global optimization.

GENERAL FLOW BY OPT LEVEL

OPT=0

The controller calls the bridge to process the output from the front end.

The bridge accumulates the IL instructions for a statement and then calls CG\$PAS in PROSEQ to process it.

[[] PAGE BREAK-- end 5-9

PROSEQ is the sequence processing controller, and it calls SBB (in SQZ), ESR (in PROSEQ), MCG and HII (in CGTM). It then resets a few flags and exits to the Bridge.

When the end of file is encountered the Bridge exits to the controller (in the host) who then does the end processing & calls the assembler (CGIA). Finally the controller exits the overlay to the host batch controller.

OPT=1

The basic flow of control in OPT*1 is similar to that of OPT=0, except that

a. The Bridge accumulates longer sequences of code for PROSEQ to process. (Extended Basic Blocks)

B. When a -well behaved- inner most loop is encountered the Bridge will attempt to collect the whole loop in memory, and PROSEQ will call COL (in GPO) to optimize it.

OPT=2

The controller calls the Bridge to process the front end output and accumulate basic blocks in TXT & control flow information in CFT.

It then calls PROSEQ who then calls SQZ and PBB .

PBB calls CMR to chain the memory references in the block into UDT, and it then saves the block on a random file on mass storage.

This process continues until the entire file has been processed.
The Bridge then exits.

The controller then calls GPO to optimize the IL and generate code for it.

Upon return from GPO the controller calls the end processor and the assembler.

[[] PAGE BREAK-- end 5-10

6.0 COMDECKs

Because CCG is assembled and executes in the environment of its host compilers, it is dependent on them for certain comdecks.

The decks and the compiler/CCG build environment are discussed in Chapter 9 of the Interface Specification Appendix.

The following comdecks reside on the CCG OPL (UPDATE OLD PROGRAM LIBRARY - a source code management tool)

CCOMGCM General Compiler macro definitions for LXQ, RPVOEF, RPVFWA, LISTL and NUPAGE.

CCOMRPV Prototype version of the RPV package and associated subroutines.
 See section 7.4.1 of the Interface Specification Appendix for a description.

CGHCRMD Code Generator Host Compiler Required Macro Definition.
 Contains the definition of macros that the host must have in the assembly text that is used to assemble the COMPS file if the C option is selected. It contains definitions of the USEBLK, RJT, ORG, REPI & SUB macros.

FA=DEFS macro definitions for the 7000 (SCOPE 2) I/O routines that parallel the -MACE- I/O routines (READW, WRITEW, etc).

C0MADEF Structured Field Definition Macros.
 A set of macros to facilitate the definition of part-word fields. These macros are used extensively in CCG and the texts.

OPRDEFS CCG Internal Language Instructions Definitions.
 OPRDEFS defines of all the instructions used in the IL in the form of a macro call with the format:

NAME OPR (properties as a string)

The deck also contains other macros that are used to process the parenthesized list of instruction properties.

In most cases OPRDEFS is used to setup an opcode processor address Jump table.

In CGTM it is used to build the descriptor table.

A listing of OPRDEFS may be found in CCGTEXT and CGTM.

PSODEFS CCG Pseudo Operation Definitions.
 PSODEFS contains macro call lines of the form:

opname P00 comment,

for all the pseudo instructions in the IL. The deck is listed in CCGTEXT & CMPLTXT where it is used to define the opcode values of the pseudo instructions.

[[] PAGE BREAK-- 6-1

CCGDBGM CCG Debugging macros. A collection of macros used in test mode to debug CCG.

PRINT - similar to a FORTRAN print statement. It calls OUTPTK to

print out values in A, I, O or Z (octal with leading space fill) formats. X, T and asterisk delimited string formats are also available.

The PRINT macro calls SVR=, RSR= to save & restore the registers unless the Nosave param is specified.

TRACER - the TRACER macro defines symbols that activate the appropriate trace macros in the code. For example, a TRACER XY will activate the TRACE XY, TBL (i.e. cause it to print snaps). This allows TRACE and PRINT calls to be placed at appropriate points in the code and activated as necessary when debugging.

SNAPRL - calls DMPRLST to print out a table of instructions in a mnemonic readable form.

DCALL - calls an arbitrary routine. The parameter list is setup in FORTRAN Aplist format. The registers are saved before and restored after.

SWAPT - prints out the contents of a table with pointers 0.TBL and L.TBL in octal.

CCGILFD CCG Internal Language Field Definitions and Table Formats.

This comdeck defines the symbols and macros that are necessary to manipulate the IL instructions and associated tables.

CCGILFD calls OPRDEFS and PSODEFS to define the opcode values.

It is called by CCGTEXT and CMPLTXT.

[[] PAGE BREAK-- 6-2

7.0 TEXTS

There are two systexts on the CCG OPL, CMPLTXT and CCGTEXT.

CMPLTXT is a subset of CCGTEXT and is for the use of the host compilers.

CCGTEXT is used along with CPUTEXT to assemble CCG.

7.1 CMPLTXT

Contains:

OPTIONS - Installation dependent EQUs

COMAOEF - Describe, define macro definitions

CCGILFD - CCG IL field definitions

and macro definitions for the basic table manager macros ALLOC and ADOWRD

7.2 CCGTEXT

Contains:

OPTIONS

COMADEF

FA=DEFS - File action macro definitions

CCOMGCM

COUNTS - Count number of names in a micro string

MX+X - Select max of 2 integers

MX-X - Select min of 2 integers

CALL - Call a routine

ENTRY. - Declare a word of data as an entry point

EQENT - Equate symbols and declare as an entry point

MOVE - Move a block of data

PLUG - Modify code during execution
 ROUTINE - Define local entry/exit line
 SETCORE - Set a block of memory to a given value
 SETZERØ - Set a block of memory to zero
 CCGDBGM - CCG debug macros
 OBG=MAC - Interactive and Batch debug package (IDP PPU dbg program) macros.
 CCGILFD - Describe defines for the following structures:
 ML. - Mod list format
 T. - Temp equivalence table format
 LC. - Label change table
 WC. - Word C of the symbol table
 The following macros:
 WRITEP - Write a pseudo op word to SLIST
 ADDWRD - Add a word to the end of a managed table
 ALLOC - Allocate table space
 PROCESS - Define a processor address
 EXT - Declare names of externals with an equivalence sign appended
 TABLES - Declare names of dynamic tables referenced in a routine as external

CCGTEXT also calls the host supplied comdecks HCDEFS and SYMDEFS to define the host dependent symbols and the host

[[] PAGE BREAK-- 7-1

8.0 CCG DEBUG MODE PRINT ROUTINES

In test or debug mode the following FORTRAN coded routines are used to print out various tables and structures in CCG in a human readable format.

8.1 ROUTINES

DMPIIT Prints out the IIT (Integer polynomial increment information table) that is formed by GPO during loop optimization. It also contains the secondary entry point PRNTRXI which prints out the region exit information formed by GPO/FXI.

DMPRLST Prints the contents of a table (TXT, SEQ, MOO, etc.) that contains the IL instructions in the 4 word per entry format.
Any words that occur after a EOQ are printed in octal•

DMPSIT Prints out the successor index table formed by BDT.

DMPTREE Print out the dependency tree prior to reformatting it as a successor index table (BOT/CIP).

DMPUDI Prints out the symbol classes in UDT prior to reformatting by UDT/AUT and the OPC names in the symbol table.
This print out is the first to appear. IT is useful when one has to look at octal fields which contain IH's.

PRNTGRA Called by GRA to print out RCT (register candidate table), RAT (register assignment table) and the values of miscellaneous flags and cells.

PPNTMIO Prints out the tables (BIT, BST, BLK) or parts of tables that MIO modified.

PRNTUDI Prints the bits set in UDT during global optimization (GPO and GRA).

PRNTABV Prints the value of a single bit vector.

PRNTBV Prints out the USE, UBD & OEF bit vectors formed by GPO.

The entry point PRNTLX prints out the live exit bit vectors after FBV has formed them.

PPNTRLI Prints a single IL instruction called by OMPRLST, PRNTMCG, etc.

[[] PAGE BREAK-- end 8-1

PRNTMCG Prints out the issue candidate list, the functional unit available times, the issued instructions in the order of issue and timing information, and the R-number in the registers. PRNTMCG also performs various consistency checks and they can usually pinpoint errors in the MCG logic.

8.2 EXTERNALS REFERENCED

OUTPTK (PRINT statements) is a special version of the FORTRAN output routines.

It combines the logic of OUTPTC and KODER. It also contains a special feature called line continuation mode which is discussed below.

It contains all the facilities of the standard package except E, F, G, V & ='s format descriptors.

SETST routine in OUTPTK to set or clear the subtitle line.

REMARK same as FORTRAN library routine which sends a message to the Dayfile making it visible to the console operator.

SYM given a IH, this function returns the DPC name in the 7I format.

GETNSYM used by DMPUDT to obtain the number of symbol table SYM.

8.3 ROUTINE STRUCTURE AND CODING CONVENTIONS

The basic structure of the routine is

SUBROUTINE card
Storage declarations
Statement function definitions
Executable code and formats
ENO line

The FIELD statement function, FIEL0(X,B,L) extracts a field with rightmost bit B (0 to 59) and length L from a word X.

It is basic, since all other part word field statement functions are defined in terms of it. For example the RF field in the R2 word of a IL instruction has the following definition!

RF(X) = FIEL0(X,36,18).

The name of a statement function which extracts a field is the same as the name of the field in the DESCRIBE, DEFINE.

[[] PAGE BREAK-- end 9-2

9.2 INITIALIZATION OF CGIA PROCESSING

Initial processing activates CGIA for the processing mode required by the Host Compiler and performs other functions:

- A. Returns to the main compiler control immediately if none of the options listed below has been specified by the Host Compiler:
 - CGIA output should be binary object code in LGO file (CO\$B = 1).
 - CGIA output should be COMPASS lines in COMPS file (CC0\$C = 1).
 - Object code listing is required (CO\$L0 = 1).
- B. Internal controls are activated to set C5IA in one of the modes as instructed by the HOST:
 - Object code listing required only
 - LGO output, no object listing
 - LGO output with object listing
 - COMPS output.
- C. Writes the end-of-record on SLIST and then the file is rewound for input reading. The first block of SLIST entries is read into the input buffer IBUF and the Buffer Index is set to point at the first word.
- E. If an end-of-file was encountered on the read, the compilation is terminated. Otherwise the main control loop (GNIW) is entered to start processing of SLIST instructions and directives in IBUF.

[[] PAGE BREAK-- end 9-3

9.3 MAIN CONTROL LOOP (GNIW) AND INSTRUCTION PROCESSORS

GNTW reads the first word of a current IBUF entry as pointed by the Buffer Index. The word is checked for its OP code value which determines what part of CGIA should be entered.

- A. A pseudo-instruction processor if the current IBUF entry is a pseudo-instruction (directive). To specify a correct processor for the particular pseudo-instruction, the OP code value is used as an index into a table that contains addresses of all pseudo-instruction processors.
- B. A machine instruction processor if the current IBUF entry is a machine (executable) instruction. Again, OP code value is used as an index into a table of addresses for all machine-instruction processors.
- C. A Process Macro Reference (PMR) local routine if the current IBUF entry is a macro reference. PMR is entered for any macro reference.

Once control has been transferred to the appropriate instruction processor or to PMR, these are responsible for reading subsequent words that belong to the current SLIST entry in IBUF (if entry consists of two or more words). The GWIW macro is always called to get subsequent words from the SLIST file. It takes care of incrementing the buffer pointer and refilling IBUF when necessary.

When processing of an IBUF entry is completed, the instruction processor or PMR returns to GNIW to process another entry. The only exception is the ENO Pseudo Instruction Processor that returns directly to the Main Compiler Control.

9.3.1 PROCESSING OF INITIAL PSEUDO INSTRUCTIONS

Initial Pseudo-Instructions are directives generated by the Host Compiler Front-end and Bridge. They may appear in any order except the USEBLK which must be the last one. Two are always required - IDENT and USEBLK? others are optional. The following functions are performed by Pseudo-Instruction Processors.

A. LCC (Loader Control) Processor allocates a space for LCC table in the dynamic working storage. Then loader directives are transferred from IBUF to LCC table.

B. IDENT (Program or Subprogram Identification)

Processor transfers the following information to the local table IDT (77 table I:

- the program name from the SYM table entry, word A
- date and time from the global cells HO\$OATE (2 words)
- control card options from the global cells HO\$CCPP (3 words)•

[[PAGE BREAK-- end 9-4

C. TITLE Processor transfers the listing title from IBUF to the global table STITL to be used by the routine that lists lines.

D. COMNT (Comment Lines) Processor transfers the comment lines from IBUF to IDT table (following the information set up by the IDENT Processor). There can be up to four words (40 chars) of comments.

E. USEBLK Processor performs various functions according to the mode of CGIA operations. For example, if object code output is required LCC and IDT tables are written to LGO. Then the PIOL table is formed and also written to LGO. If COMPS output is required, COMPASS statements for IDENT, COMNT and other directives are written to the COMPS file.

Other functions performed for any mode are the space allocation for a Link and Fill Chain Table (LAFT), setting the table overflow exit in the Dynamic Table Manager, and initialization for the Main Control Loop GNIW.

The latter involves initial setting of the ●●current block number-in-use- (as defined by USEBLK directive), zeroing-out the origin counter (ORGC) and forcing the count of available parcels (NPR) to 4.

Note that the block number of a current local or common block-in-use and current values of ORGC, NPR are kept in a local variable AI.

9.3.2 PROCESSING OF MACHINE INSTRUCTIONS

Unlike the pseudo-instruction processing, there are common processors for groups of similar instruction:

- A. Processor for XIBJXX instructions (any register instruction whose RJ operand is a B register, such as ILS, IRS, NR, RNZ, VP, PKI).
- B. Processor for XIXJXX instructions (all arithmetic and some Boolean instructions)•
- C. Processor for XIXKXJ instructions (remaining Boolean instructions STR, IMP, EQV)•
- D. Processor for XIXJ instructions (XMT, XMTC, CS).
- E. Processor for shift and mask instructions (KLS, KRS, FMA).
- F. Processor for Read/Write LCM instructions (DRL, DWL).

Their function is to recognize each instruction by its OP code and get the numbers of machine registers assigned to operands. Each processor exits to a common routine TYI1 that completes the instruction processing by calling WRTEXT to place the binary in the current text table and format a line for the object listing if selected.

[[] PAGE BREAK-- end 9-5

9.3.3 PROCESSING OF MEMORY REFERENCE AND REGISTER SET INSTRUCTIONS

There are several common processors entered for groups of similar instructions:

- A• Processor for the 15 bit set instructions (SLD, SST, SA, SDL, SDS and SS).
- B. Processor for the 30 bit set instructions (LD, ST, STT, PLD, PST and S).

Their function is to get the OP code value, types and numbers of registers assigned to operands and, if applicable, values of CA (Constant Addend) and IH (sym table ordinal and pointer to the corresponding table entry). Then each processor exits to a common routine PLD2.

PLD2 completes the instruction processing as follows:

- Final OP code is generated from the input OP code value with regard to the type of registers used for instruction operands.
- If the final instruction is one-parcel (short set or load/store with address operands in registers only), PLD2 exits to TYI1 to complete the processing, otherwise the final relative address is computed by adding the CA value (if non-zero) to the relative address (RA) of a symbol indicated by IH and WRTEXT is called to output the binary & format the object listing.

9.3.4 PROCESSING OF JUMP INSTRUCTIONS

There is one processor for each Jump instruction JIN, JPBB, JPX, RJ3, UJP, RJ6 and RJXJ. Basic functions listed below are similar for each processor:

- Final OC code value is computed and a relative address (RA) of a destination label is fetched from SYM table (if source label) or from GL table (compiler generated label).
- Final relative jump address is computed as
 - $K = RA$ for JIN, JPBB, JPX and an RJ part of RJXJ
 - $K = CA * RA$ for RJ3, UJP
 - $K \ll 0$ for an RJ part of RJ6
 - $K = RA + RA(HC\$RJ TBN)$ for a lower part of RJ6 where!
 - CA = value of CA field from Jump SLIST instruction
 - $RA(HC\$RJ TBN)$ = value of RA for a symbol defined by the Host Compiler in HC\$RJ TBN.
- Processor gets the register numbers assigned to register operands RI, RJ (where applicable).
- The binary instruction(s) associated with the Jump type is generated!

JP BJ+k for JIN; BJ is specified by RI

[[] PAGE BREAK-- end 9-6

JT1 BJ,BI,K for JPBB; BJ is specified by RJ and BI by RI

JT2 XJ,k for JPX: XJ is specified by RI

RJ K for RJ3

EQ K for UJP

NJT2 XJ,»+1 for RJXJ XJ is specified by RI

RJ K RJ 0 for RJT; CA is taken from CA field by RJ6 12/CA,18/K

where!

JT1 is one of the Jump conditions EQ, NE, GE, LT

jt2 is one of the Jump conditions DF, ID, IR, MI, NG, NZ, OR, PL

WRTEXT is called to output the binary and format a line for the object listing.

FPU is called to force the position counter to the beginning of the next word after the JIN, UJP & RJ3 instructions.

9.3.5 PROCESSING OF STANDARD PSEUDO INSTRUCTIONS

This paragraph describes processing of standard pseudo instructions (directives) except the END pseudo and those that belong to the DELIMITER initial group described in section 9.3.1 or to the macro pseudo instruction group (see section 9.3.6).

USE and USELCM

Calls CUB to do the block switch, then it selects the correct DPC opcode & calls LPO to format it. Finally it calls WRTEXT to list the line.

8SS Calls SLP to substitute any local (macro) parameters and list the line.

Then it calls FPU to force upper. If a label is present, then it gets its address and checks it against the current value of the org counter. If they are not the same it puts a message in the object listing and increments NAE, the number of assembly errors. If the BSS is not from macro expansion it is listed. Finally DTT is called to flush out the current text table and a new text table is setup with the org counter advanced by the word count of the BSS.

LAB The LAB pseudo is simply a BSS in a different format. It is converted into BSS format and processed.

BSSZ Reserve WC locations initialized to zero. Initial processing is similar to the BSS (param substitution, force upper). In order to minimize the size of the binary file, the processor outputs zero words to a text table if WC is less than 4, and one or more REPI tables in the other case.

[[] PAGE BREAK-- end 9-7

CON Set the flag FMCA to a so that the object listing is 'CON SY*CA'. Call SLP to substitute parameters. Restore FMCA to a comma. Form RA[sym]+CA and call WRTEXT to output it. Call AXR to add the relocation information to the text table.

OATA Input to the data processor consists of a word count (WC) followed by WC data words. In the case WC is zero, the low 18 bits contain the data item. In the WC non-zero case the processor goes through a loop to get, format & output each word.

DIS The input format is similar to DATA. The processing loop consists of inputting up to 5 words, formatting a line "DIS 0, words" outputting the binary and the object listing line.

HOL The HOL Processor performs one of the following functions:

- If COMPS output is required, a character data declaration is written to the COMPS file to represent one of the following constants:

nH "string" if FF parameter in HOL SLIST pseudo if H

nL "string" if FF is L

nR "string" if FF is R

where n is the number of characters in "string", "string" is a string of characters from the next SLIST word following the first word of HOL SLIST pseudo.

- If object code is required, the character constant defined by "string" converted to the binary form and WRTEXT is called to output it.

ORG The ORG pseudo in CCG is a macro which is defined in the comdeck CGHCRMD. A complete description of its function is in 6.5.2 of the Interface Specification Appendix. The processing of an ORG directive is straight forward in that CUB is called to switch blocks and SOC is called to set the new value of the ORG counter.

REPI The REPI Processor performs one of the following functions:

If COMPS output is required, REPI COMPASS pseudo is generated from information supplied by

the REPI SLIST pseudo. Destination location of the replicated code is defined by the preceding ORG SY,WC pseudo.

[[] PAGE BREAK-- end 9-8

- To output the binary, the REPI processor completes the current word in the text table (forces upper) and dumps it to LG0, It then resets the text table header word and writes the REPI table directly to LG0,

DCS The DCS Processor function is to get the micro name specified in the next SLIST word following the first word of DCS pseudo and to transfer it to a Micro Name Table (MIC). This table is used by a macro processing facility of CGIA.

9.3.6 PROCESSING OF SLIST MACRO REFERENCES

An integral part of CGIA is a macro processing facility that provides the means to process SLIST references to user defined macros. The external definitions of the facility is contained in section 6.6 of the Interface Specifications Appendix.

The complete macro facility consists of -

MACROS The static user macro definitions

PMR The macro reference processor

GNMW The main loop macro expansion & the macro only pseudo op processors (BTW,ACI,ARI,IF.XX,SET,MIC, etc.)

Here we will discuss the last three parts.

PMR (Process Macro Reference) is called by the main loop (GNIW) when a macro reference is encountered. PMR reads the macro words and unpacks the actual arguments into PBUF, one per word, which is used by the other processors to reference them.

Next the text of macro expansion is moved from MACROS to the macro input buffer, starting at 0.MXB. An EOM (end of macro) opcode is placed at the end of the expansion to force return to the macro processor. Then PMR substitutes the parameter values in the expansion for the specified words.

If the object list flag is on FMC is called to format and list the macro call.

Finally A5 is set to 0.MX3-1 and a Jump to GNMW1 starts the macro expansion.

Note that the flag MIW is non-zero during the expansion, & serves as a no-list flag to LPO for the pseudo ops in the expansion.

When the EOM opcode is encountered, the processor restores the SLIST Input buffer pointer, clears MIW and exits to GNIW.

The main loop for macro processing at GNMW is very similar to GNIW.

It fetches the next input word from the macro expansion buffer & uses the opcode to Jump to the appropriate processor.

On return GNMW checks the instruction address to see if the previous instruction was the last instruction in a BTW/ETW pair, and if so calls WRTEXT to output the binary.

[[] PAGE BREAK-- end 9-9

BTW Loads An initial "Template Word" from a location specified by BTW Pseudo into the local variable VFDW.

The word will be modified by other pseudos in the BTW/ETW range to form a final binary word that will be written to LG0. Then an address of ETW Pseudo is computed and saved in VFDW +1 word. It is used by GNMW to terminate processing of the current BTW/ETW range.

ACI Gets the constant value CON defined in the BIT-LEN low order bits of an ACI Pseudo. The value is added to a BIT-LEN long field in the current VFDW word starting at a TOP-BIT position. (Note that TOP-BIT, BIT-LEN, CON are parameters of the ACI Pseudo.) If the ACI Pseudo specifies one of the symbols U, M, L instead of values TOP-BIT, BIT-LEN, the ACI Processor interprets this as

TOP-BIT=47, BIT-LEN*18 for U
TOP-BIT=33, BIT-LEN=18 for M
TOP-BIT=17, BIT-LEN*18 for L

API ?? Gets the RA value of a symbol identified by SYM field of an ARI Pseudo. The value is added to a BIT-LEN long field in the current VFDW word starting at a TOP-BIT position. The processor also indicates that the field has a relocatable address value by setting corresponding bits in the relocation test word. If the ARI Pseudo specifies one of the symbols U, M, L, it is treated in the same way as the ACI Pseudo. Note that each group of 15 object text words written to LG0 is accompanied by an address relocation word that indicates locations of program-relocatable address information within the group.

ASV Gets the micro name pointed to by a MIC parameter of an ASV Pseudo. The name is fetched from MIC table. (See DCS Processor or a MIC Pseudo Processor that sets MIC entries.) The name is "logically added"

(OR operation) to a NC*8 long field in the current VFDW word starting at a TOP-BIT position. If the micro name is shorter than NC characters, it is extended by blanks to the right. If it has more than NC characters, it is truncated on the right side.

IFT Tests the parameters P1.P2 for a condition specified by the parameter IF.XX. If the condition is met, the processor returns to GNMW with no further processing, so the macro pseudos following the IFT Pseudo are processed by GNMW. If the condition is not met, a pointer into the macro buffer 0.MX8 is advanced past a corresponding ELSE or ENDIF Macro Pseudo, so the macro pseudos in the IFT/ENOIF or IFT/ELSE range are skipped,

[[] PAGE BREAK-- end 9-10

ELSE Advances the 0.MXB pointer part to the corresponding ENOIF Macro Pseudo, so the macro pseudos in the ELSE/ENOIF range are skipped by GNMW.

Note: no processing is done for ENOIF by GNMW as it serves to terminate an IFT/ENOIF or ELSE/ ENOIF range.

ETW Pseudo has no processor since its processing is done by the main loop GNMW. It consists of transferring the completed binary word from VFDW variable to LG0.

MIC Processor makes a micro name from a symbolic name identified by the SYMORO parameter according to parameters FC (First Character), NC (Number of Characters) & SC (Separator Character).

For instance, if FC=NC=SC=0, an entire symbol name extended by blanks to 8 characters (if necessary) is used to make a micro name. See Interface Specification Appendix,

paragraph 6.6.2.3, for the set of rules on how to use FC, NC, and SC parameters.
The micro name is then stored into the MIC table entry identified by the MI parameter.

SET Gets the operands identified by the arguments 01,02 of a SET Pseudo.
Each argument can identify a SYM table symbol (in which case the operand value is a relocatable address value RA), a constant, a local symbol defined in MACRO S module, DCSS symbol or a current ORGC value (*).
See Interface Specification Appendix, paragraph 6.6.2.3 for more details.
An operation identified by OP code parameter is performed upon the operands and a result operand stored to a location specified by an X parameter.

9.3.7 END PSEUDO PROCESSING

It is performed by the ENO Pseudo Processor. It involves the following functions.

- If any errors are encountered by CGIA, an error message is written to the standard output listing file.
- If COMPS output is required, an END COMPASS line is written to COMPS file followed by EOF record & the COMPS file is closed.
Then the ENO Processor returns to the Main Compiler Control.
- If an object listing is required, ENO COMPASS line is written to the std output list file.
Then the ENO Processor returns to the Main Compiler Control if object code generation is not required.

[[] PAGE BREAK-- end 9-11

- If object code is required, the ENO Processor generates the final LINK and FILL loader tables by sorting the LAFT table. They are written along with the XFILL table to LGO file. In addition, if the END Pseudo specifies an ordinal of a symbol for the program transfer entry, the symbol is obtained from SYM table, XFER loader table formed and written to LGO. Finally EOR is written on LGO and the ENO Processor returns to the Main Compiler Control.

Notes:

- LAFT table as generated during CGIA processing, represent fill loader bytes for referenced variables in common blocks & link loader bytes for referenced external symbols.
The ENO Processor calls DAT to sort & format it as FILL and LINK loader tables.
- If the END Pseudo defines a label symbol (in CA field), it is placed in the label field of the ENO COMPASS line generated to COMPS or output listing file.

9.4 THE OBJECT LISTING

When an object list is selected (HO\$LO\$O) APT (WRTEXT) is adjusted so that the listing processor is called from it after the current instruction is placed in the text table. The main listing processor, FBD (Format Binary Data), is responsible for formatting and outputting a print line in a format similar to COMPASS. In terms of FORTRAN print and format statements it would be described by:

```
PRINT 10, ADDRESS, BINARY, RELOC, LINE
10 FORMAT(3X,06,1X,020,A10,8A10)
```

Where ADDRESS is only printed at the beginning of a new word, etc.
FBD calls FMI to format the SLIST instructions as a OPC line image if this has not already been done.
It then converts the binary to octal, gets the relocation, adds the binary address and outputs the line.

The rest of the macros and subroutines are discussed in their listing order.

BFN A macro to blank fill a string of chars in X6 & the bit count is in B3 (6 times N.chars).
Result register is the argument to the macro.

ADC Adds a string of up to 10 chars to the current line Image which is being built in the buffer LBUF.
On entry to LBUF A7 points to the last completed word in LBUF, X7 holds the current word (partially filled) and B4 the unused bit count for X7.
The caller sets the string to be added in X6 in zero L format and the bit count in B3 and calls ADC.

[[] PAGE BREAK-- end 9-12

TSB Terminate String Buffer. Called to terminate the current line & write it to COMPS (C option).

CMA Convert Macro Argument to display code. CMA is called with two arguments, the parameter value and its type code (see SMACRO definition in MACROS). It uses the type code as an index to Jump to the appropriate processor, which converts the value to OPC and exits with it in X6 & the bit count in B3.

CON Convert Octal Number to display code. Convert a number less than 24 bits long to octal display code in the format zero L:nnnB. The colon is used so a minus sign or comma can be placed before the number.

FMC Format Macro Call. Called by PMR to convert the macro call to display code, it first checks to see if the first argument is to be placed in the label field, and if so converts the argument and places it there.
It then adds the macro name to the assembled string and loops to convert the arguments and add them to the string. Finally it calls TSB to terminate the line and exits.

CDO Convert Data to Octal. Given a fill word 5A00000 or 10H0---0, it converts the binary digits to octal and adds them into the fill word.

CFW Convert Full Word. Does an 020 format conversion by calling CDO twice.

CRI Convert Relocation Information. Given the RL field in RBW0R0. CRI set up 10 blanks (ABS), 2A+ (program relocation), the common block name or 10H<ext>.

FMI Format Machine Instruction. Given the binary and the number of parcels (1 or 2), FMI decodes the instruction and places a COMPASS instruction in display code in LBUF.
The algorithm is a fairly straight forward conversion that uses some tables to save space.
The routine is a modified version of the one in the TS version of FTN 4.

9.5 LOADER TABLE OUTPUT ROUTINES

The preliminary binary output, 77, PIDL, EMTR and LDSET tables are put out by the USEBLK processor.

The LINK, FILL and XFILL tables are output by the ENO processor. In the case that table manager needs space it may call DAT to dump them. All writes to the LGO file are done via the WRLGO macro which calls WWL. A "TRACER LGO" will cause every write to LGO to be printed. APT is called by most of the processors to place up to 4 parcels of data in a text table. It calls FBD to print the object listing and ATR to add the relocation information to the text table under construction.

[[] PAGE BREAK-- end 9-13

ATR Adds the text relocation to the text or link and fill tables
 for 18 bit address's that occur in the standard positions only.
 For program relocatable address's the relocation information is added directly to the current
 text table. For common & external relocation the appropriate information is saved in LAFT
 until it is reformatted and output by DAT .

DTT Terminates the current text table and writes it to the LGO file.
 The caller is expected to setup the header word (RB and ORGC) of the new text table.

FPU is called to "force upper" after an unconditional Jump ... If the current word is non-empty
 and not full, then it fills the remainder with no-ops and stores it in the text table.
 If the text table is full, it writes it to LGO and sets up a new one.

AXR Add Extended Relocation is called to place non-standard relocation information in the binary
 file. Includes address fields that are not 18 bits long or those that do not have a low bit
 or bits 0, 15 or 30.
 The routine first checks its arguments to see if the address field can be handed by the
 standard relocation subroutine, and if so it calls ATR to record the relocation.
 Since the format of the relocation word is the same for all types we set it up, and in the
 common/program relocatable case save it in XFT. For external symbols a short XLINK table is
 written to LGO.

DAT (Dump Accumulated Tables) DAT is called by the table manager or the end processor to write
 the XLINK, LINK & FILL tables to LGO. They may appear on the binary file anywhere after the
 initial tables (77,PI0L,ENTR). The routine first writes the XFILL table to LGO. It then
 sorts LAFT into two tables, one holding the external references & the fill references sorted
 by block number. It then loops to form the FILL and LINK tables and write them to LGO.
 Finally it resets the table lengths and exits.
 DAT calls AFT and DLT to add words to and write out the LINK and FILL tables.

9.6 CGIA DATA STRUCTURES

This section describes formats of various tables and variables generated, referenced, and modified during CGIA processing. The SLIST assembler input as well as input formats of major static tables are not included since their description can be found in Interface Specification Appendix, sections 6.1-6.6 (SLIST) and 5.2 (Static Tables).

[[] PAGE BREAK-- end 9-14

9.6.1 CGIA TABLES

IDT (IOENT) Table is set up during the Initial Pseudo Instructions processing and written to
 LGO by the USEBLK Processor. It contains program name, date and time, operating system IO
 & version number. Host Compiler ID & version number, PSR level, relocation type,
 control card options and up to four comment lines.

LDSET Table is set up and written to LGO if any libraries are defined by the Host Compiler.
It contains a list of library names.

PIDL Table is set up and written to LGO if any common blocks are defined in the source program.
It contains the common block descriptors.
Each descriptor specifies the common block name and length and if the block is LCM/ ECS resident.

ENTR Table is set up and written to LGO if any ENTRY points are declared in the source program.
It contains ENTRY names and their RA values. In addition, if an ENTRY is in a common block,
the block relative address is supplied.

LBIT Table is generated by the USEBLK processor from the Host defined LBT table.
There is a two-word entry per each local block. USEBLK processor initializes each entry to

W1 = 12/PfINPR),15/0,9/IRB,24/IORGC
W2 = 60/0

where -INPR=4 - Initial Parcel Count (Packed)
 -IRB=i - Initial Block Number
 -IORGC=0 - Initial Origin Counter Value

Later, as a result of USE or ORG pseudos, the entry is updated to
W1 = 12/P(NPR),15/0,9/RB,24/ORGC
W2 = CW

where -NPR, RB, ORGC are current values of parcel count, block number, and origin counter
of the local block whose usage is being suspended by USE or ORG pseudo.

Note that NPR, RB and ORGC for a new "block-in-use" are loaded from LBIT entry of such block into
the local variables AI (1st word) & OW (2nd word). See next paragraph for more details
about AI,CW.

LAFT (Link & Fill) Table is initialized by the USEBLK processor (space reserved).
Then an entry is made each time an object code instruction referencing an external symbol
or a symbol residing in a common block is generated. The LAFT entry contains:

[[] PAGE BREAK-- end 9-15

12/0,18/LCBN or EXN,30/FILL or LINK Byte

where --LCBN loader number for a common block
 --EXN external number
 --FILL or LINK byte contains a relocation factor,
 block number or the current block-in-use and the current ORGC value.

FILL Table is generated from the LAFT table and written to LGO by the END Pseudo Processor.
(See also description of ENO Pseudo processing.)
It contains relocation information for referenced symbols in common blocks.

LINK Table is also generated from the LAFT table and written to LGO by the END Pseudo Processor.
It contains information about referenced external symbols.

XFILL Table entries are generated during CGIA main loop processing to represent referenced

variable in common blocks that reside in LCM/ECS.
The table is written to LGO by the ENO Pseudo Processor.

XFER Table written to LGO by the ENO Pseudo Processor represents a symbol for the program transfer entry.

XTXT (Binary Text) Table is used to hold the object binary code (instructions, constants) being generated during CGIA processing. When the table is full, it is transferred to LGO and reinitialized for another code block. A full table contains the following information:

- TH (Text Header Word) contains table ID, word count for the following binary text (TD words), block number of the current "block-in-use" and a current ORGC value.
- TR (Text Relocation Word) contains relocation bytes for any relocatable address references within the following binary text.
- TO's (Text Words) contain binary code.
Note that normally there are 15 text words in full TXT table, unless fewer words have to be written to LGO because the usage of the current block is suspended or terminated.
In connection with processing of some pseudo instructions, CGIA also generates and writes to LGO the following loader tables:

- XREPL Table for each REPI Pseudo
- VFDP Table for each VFOP Pseudo.

Note that XREPL table is also generated by BSSZ Pseudo

[[] PAGE BREAK-- end 9-16

9.6.2 LOCAL VARIABLES

The following text describes those variables only that are mentioned in the CGIA functional description.

AI contains information about the current block-in-use:
2/01,10/NP9,1/LCM,14/0,9/RB,24/ORGC

where: NPR -number of parcels available in the current text word CW.

LCM -if 1, the current block-in-use is a common block residing in LCM/ECS.

RB -block number of the current block-in-use.

ORGC. -current origin counter value.

CW contains a current text word being assembled.
When filled (NPR decreased to 0) or when "forcing upper" occurs, it is transferred to the text table (XTXT).
Forcing upper is done if any of the following conditions arises:

- one parcel available only for a current two-parcel instruction.

- a current instruction is RJG, RJXG.
- a new block specified by ORG or USE instruction is a common

Note that empty parcels are filled with NOP instructions when CW is partially filled block.

[[] PAGE BREAK-- end 9-17

10.0 MACROS - HOST COMPILERR STORAGE MACRO SKELTONS

MACROS contains the definitions of the host compiler's macros that generate storage reservation and initialization directives in the relocatable binary. MACROS consists of a set of static tables that are accessed by the macro expansion code in CGIA. The deck contains macro definitions whose function is described in section 6.6 of the Interface Specification Appendix, followed by a *CALL SMACROS, which is the name of the comdeck on the host's Oldpl containing the SMACRO definitions.

PBUF A 7 word buffer to hold the actual arguments (parameters) of the current macro call. The first word is always zero. The format of the other words are 36/0,24/arg value.

F\$MIC Micro string value table, 2 words per entry. Format is 60/0L string,60/6*Nchar

ORG CTR Current value of the origin counter in AD. format, setup & used by the SET. processor.

F\$MACS FWA of the SMACRO opcode index table.
The index table contains one word for each SMACRO in the format

6/,18/n.PR,18/len,18/fwa-1

where n.PR is the number of parameter references that have to be relocated,
len is the number of words in the macro skeleton and fwa is its first word address.

SS BIAS The difference between L0, the beginning of the special symbol value table and PBUF .

F\$MXB The first word address of the macro expansion buffer that CGIA/PMR moves the macro expansion to. It is contained in the common block CCG.SCR

18.2 MACRO DEFINITIONS

LITORD When a literal appears as an argument in IF/Z, SET., etc., LITORD is called to place it in the PARAMB block and give it a name so it can be referenced as if it was a macro argument.

SMACRO Defines the beginning of a storage macro definition.
First it sets up the macro information word in the block

[[] PAGE BREAK-- end 10-1

SKELS in the format

36/6L MAC-NAME,3/LAB,18/PARAM-TYPES,3/0

LAB is 1 if the first argument does not appear in the label field when the macro is listed & zero if it does. The param-types field contains a three bit field for each macro argument which is its type (CON, SYM, etc.). The macro also sets up the formal argument name as a cell in PBUF so the other macros can reference it. Finally it initializes a group of re-definable symbols that the ENDS macro will reference to set up the index word.

ENDS The ENDS macro terminates a SMACRO definition. It determines the length of the SMACRO definition, sets up parameter relocation bytes at the end of the Skelton and pointer, length word in the block INOEXT. The last is used to find the macro skeleton when the opcode is encountered by the assembler.

FPB Form a RELOC byte for a reference to a macro parameter from a pseudo op. When a CON, BSS or USE appears in a SMACRO the macro parameter values must be substituted into the pseudo word prior to passing it to the opcode processor. This eliminates special code in the processor whose arguments come from both the input stream and macro expansions. The arguments to FRB are the type of symbol (L, S, C, Q, ●●●) And the position of the bottom bit. For macro parameters it forms the symbols PR.1, PR.2, ... , whose values are 6/bottom bit, 6/index from last relocation to this relocation. The ENDS macro sets up the relocation info at the end of the macro skeleton as a series of 12 bit bytes.

BTW Begin Text Word. BTW begins the definition of a word of data whose fields may be filled in with instructions, address or string values. A BTW must be followed by a matching ETW. The argument to BTW is the address of a background word or template. If it is absent, then a background of zero is assumed.

In either case the macro forms a word

12/P(0C.BTW),18/ETWA,12/,18/TWA

where ETWA is the macro relative address of the last word in the BTW/ETW pair and TWA is the address of the template word. BTW also sets the symbol ●VFD and the micro BTW which some of the other macros use to detect errors in the SMACRO definitions.

ETW End Text Word definition. It checks for errors in the SMACRO definition and defines ETWA for the preceding BTW.

[[] PAGE BREAK-- end 10-2

ARI Add Relocation Information to a text field. The ARI macro sets up a word of the format

12/PCOC.ARI),18/59-LEN,12/P(B0T-BIT),18/SYM

Where LEN is the length of the field,
BOT-BIT the lowest (rightmost) bit and
SYM is the address of the word holding the IH of the symbol.

ACI Add Constant Information to a text field. ACI format is the same as ARI.

[[] PAGE BREAK-- end 10-3

11.0 CGTM - CODE GENERATOR TABLE MANAGER

CGTM contains the CCG Table Manager and other miscellaneous routines that are common to the code transformer and assembler phases of CCG.

Since CCG may be split between overlays, CGTM may be thought of as the cradle of CCG. In the following the routines are discussed in their listing order.

MISCELLANEOUS ROUTINES

CG\$IEP Initializes end processing. Sets a new low memory limit for the table manager, closes the OPT=2 random file & resets some other flags associated with the table manager.

PUNT Terminates CCG processing when insufficient memory is available. It increments the host compiler's fatal error counter & exits to a host supplied address.

CG\$SCT Search constant literal table CVT for previous occurrence of a constant (the input argument). Add it to CVT if none. In either case return the ordinal of the entry in the table for use in the CA field of LDC instruction. The code consists of first adding the constant to CVT followed by a search terminator loop to look for the first occurrence of the constant. On finding a match it adjusts the length of CVT and returns the ordinal in X6.

CG\$ENC Enter n-word constant literal in CVT. Similar in function and logic to SCT, except that the argument is a multi-word constant.

CG\$FCU Force use of K'th constant in CVT. FCU sets the K'th word in CUT non-zero to indicate that it is "used". The main capability provided by FCU is the ability to use constant literals in AP and 10 lists.

WPW Writes a pseudo op word to SLIST

11.1 END PROCESSOR ROUTINES

CG\$RBT Relocate Block Table. Terminate all blocks (force upper) and convert the block table (F\$LBTT) from a block length format to a block FWA. Save the sum of the local block lengths in N\$SLBT

CG\$CUB Change use block. Switches code output from one local block to another. It updates L8T and outputs a USE pseudo op to SLIST.

[[] PAGE BREAK-- end 11-1

FSU Force next sequence upper. Called by the code transformer to force upper.

CG\$EP End Processor. Called by the host. First scan CVT to eliminate unused constants from CUT, and then outputs the table (CUT) to SLIST as CON. BSS 0, followed by the table as a series of OATA words. Finally define the address's & reserves storage for the special symbols IT. & OT. .

CG\$DSA Define symbol address. Defines the address of a symbol (first argument) & output storage for it (" second ") to SLIST in terms of a BSS pseudo.

11.2 WII - WRITE ISSUED INSTRUCTION TO SLISI

WII Write Issued Instruction to SLIST. WII is called by CG\$PAS after MCG(OPT < 2) or by GP0. Its primary functions are final conversion of the SLIST instructions, address definition and updating the sub counts.

WII consists of a main loop to extract the opcode processors for the instructions that have to be inspected & a return point to adjust the parcel and block length counters. The following instructions are given special processing in WII.

NOP A NOP is placed at the end of the sequence by MCG.
Where it is encountered by Nil, it terminates processing & writes the sequence (contents of PIT) to SLIST.

DRL,DWL These instructions are processed only if level 0 address substitution is being used by the host. In this case a DRL/DWL with a non-zero IH is really a "SUB0". In this case the SUB0 count for that formal parameter has to be incremented.

TLD,TST In these instructions the CA is an index into TET, and are converted into a real CA at this time so that the compiler generated temps occupy a minimum amount of space. The code checks to see if a final assignment of a CA has been made, or if one has been assigned (OPT=1), assigns one and in either case adjusts the CA field in the instruction.

ILD Processed as a TLD if the IH is IT., else processed as a LD.

LD,ST,STT If IH & CA are zero, then process as a one parcel instruction, else extract IH & if I is zero then set the MAT bit in symtab for this symbol so that the host will issue storage for it.

If IH is a formal parameter, then CSR is called to count the SUB reference. Finally, if there is an H2 word, then it is skipped over.

[[] PAGE BREAK-- end 11-2

LOC The CATH entry in CUT is set non-zero to indicate usage of the constant.

LDV CG\$AV0 is called to assign & return a final CA & the instruction is converted to a LD.

JPX,JPBB Call RLV to check for a change of IH in OPT-2.

JIN,UJP Call RLV and then force upper.

RJ3 Force upper after.

RJXJ Force upper before, increase the block length by 1.

RJ6 Force upper before, increase the block length by 1, & count a SUB ref if the IH is a formal parameter.

LAB Force upper before & define the address of the IH as the current block length.

CSR (Count Sub Reference) Gives an IH which is a formal parameter.

Up the sub count field in FPI & sets the subs encountered flag non-zero. (CC\$SUB)

RLV Checks a referenced label value for substitution in (OPT=2).
The label will be changed if the LC bit in word B of the symtab entry is on.

CG\$AVO Given an index into the Vardim table (VDI) check to see if there has been a previous use of this entry, and if so returns the CA value.
Otherwise it assigns the next available value (N\$VD) as the CA,
marks the entry as used and increments N\$VD.

CSN Converts an IH to its DPC name in zero L format.

MISCELLANEOUS ROUTINES

/OPRDEFS/ A common block that is used by PRNTRLI in debug mode to print out the IL Instruction opcodes.

F.RDT (F\$RDT table of IL instruction descriptor words. OPRDEFS comdeck is used to form it.

ISC= Initialize Small Core. called by SETCORE/SETZERO macros to initialize memory blocks of \

SST Fancy version of SHL, it is a shell sort of a single word/entry table that allows a key mask and shift when sorting.

SHL A simple shell sort for single word/entry tables.

[[] PAGE BREAK-- end 11-3

11.3 TABLE MANAGER

The CCG table manager consists of the table vector comdecks (CCGTMTV, etc.) and the table manager routines which consist of ATSS\$, ADW\$, MTU\$, AFT, CWS, the table overflow processing routines (TOV, TOB, MOREFL, AST) & the test mode table dump CCG\$PTC).

The table manager was adapted from an early version of the MACE table manager, and was revised and expanded to meet the needs of FTN 4.x and CCG. The algorithm used to reallocate the dynamic table when one table overflows is Garwick's, described in Knuth, Vol #1.

The CCGTMTV comdeck has of the B\$TBL macro definition to setup the table vectors FTAB & LTAB, the basic list of tables used by CCG as TABLE macro calls, followed by calls to the host supplied comdecks CGHCDTD and CGHCSTD.
The FTA8 vector holds the FWA's of the tables & LTAB vector their lengths.

ADW\$ called by the ADDWRD macro (in CMPLTXT) to add a word to the end of a managed table. If there is not enough room between the tables it calls ATSS\$ to get space. In either case it then adds the word to the end of the specified table.

ATSS\$ is called by the ALLOC macro to allocate n extra words for a table. If there is sufficient space between the requested table and the one after it, it updates the length & exits. If a reallocation is required, it saves some registers & computes the remaining table space, which is the length of the dynamic table area minus the sum of the lengths of the tables.

If there is sufficient space between the tables for the increase, then the tables are packed down to low core and then reallocated and moved to their new position. Finally the registers are restored and AT\$ exits to the caller. In the case insufficient space is available, AT\$ lumps to AT\$8, which calls one of the table overflow processors (TOV, TOB, etc.), which attempt to get more space. A successful return from one of the overflow processors is to AT\$9, an unsuccessful return is to PUNTE.

AFT called by GP0 to activate the first table (BLK) - used during loop optimization.

MTU\$ moves all the tables up to the top of memory (RA+FL), moving the last table first.

CWS called by CG\$INIT to determine how much FL is available & what the working storage limits are.

[[] PAGE BREAK-- end 11-4

TOV, TOVA and TOB (the table manager overflow routines) called by AT\$ when there is insufficient space to expand a table. They attempt to get more space by first trying to shrink tables that are in memory, & then if this isn't enough they call MOREFL to increase the FL & move the static tables up.

MOREFL makes a MEM call to the system to increase the compiler's field length. It then calls AST to move the static tables up.

AST an internal routine called by MOREFL to update the pointers to the static tables and move them up.

CG\$PTC is called by the host's compile time reprieve processor in test mode to print the contents of the compiler tables in octal & a formatted dump of the current IL instruction sequence in TXT.

[[] PAGE BREAK-- end 11-5

12.3 MIO - OPT=2 MASS STORAGE [PAGEING] I/O ROUTINES

General Overview

Routines handles dynamic storage allocation for extended basic blocks during OPT=2 processing. Blocks are paged from LCM/ECS or the disk file ZZZZOP into an SCM table (page buffer) BLK as they are requested.

The random index words in BIT (block address and status table) and the block address words in BST (list of active blocks) are updated for each block that is moved.

They contain current positional information including the SCM first word address of the block text, disk address, LCM/ECS address, length, and status bits for each block.

These routines are also used in a more limited manner to allocate block storage for OPT=1.

This type of simulated virtual storage is necessary because the program text blocks are processed by the global optimizer in a non-sequential manner and it may not be possible to keep them all in core at one time. It is also used to read and write overflow graph tables to the disk file ZZZZOP.

MIO allocates space for blocks in the table and in LCM/ECS using Knuth's boundary tag method. This technique is described by Gries in Compiler Construction for Digital Computers section 8.10.

It requires header & trailer words for every block that are used to keep track of free & allocated areas of space.

The advantage of this method is that essentially a fixed amount of time is necessary to free an area and collapse it with adjacent free areas if possible.

Other methods require some type of list search to identify adjacent free areas.

The format of each used and free area is shown in the diagram below. The header word indicates if the area is free or currently allocated & the length of the block (including header & trailer words). For allocated areas it also contains BIT and BST table indices for the block and a set of status bits giving the purging priority final write flag, and holding block indicator. The trailer words are used to keep a doubly linked list of available space. The forward link field points to the next area as the list, while the backward link field points to the previous one. The list is terminated by a zero link field. The use of header and trailer words also simplifies garbage collection procedures.

The following diagram illustrates storage allocation for three blocks.

One is a holding block which was created during optimization and consequently has no copy on disk or in LCM/ECS.

The other two show how the address type bit in the random index word is used to determine whether the out-of-core copy of the block is in LCM/ECS or packed into a disk record.

It should be noted that table entries referring to the FWA and length of a basic block in BLK pertain to the text of the block and do not include the header and trailer words.

This is because the other routines of the optimizer that use these table entries do not allow for the two extra words per block. However, the pointers used with blocks on disk or in LCM/ECS are to the header words

since these are internal to MIO. The diagram also shows how the linked list of available space is structured.

The storage allocation illustrated is for basic blocks in the SCM table BLK.

[[] PAGE BREAK-- end 12-1

BIT * Block Information Table

	IC	AT	LEN	OFS	FWA	RA	LCM
	*	-----	*				
	*	-----	*				
(11)	1	1	54		42102	. 320	
	*	-----	*				
	*	-----	*				
(13)	0	0	100	22		112	
	*	-----	*				
	*	-----	*				
(13)	1	0	24		42262		
	*	-----	*				
	*	-----	*				

BST - 8lock Status Table

	BLK	PRI	HB	FW	BI	LEN	FWA	
(0)	*-----◆							
	1	1	1	0	17	. 24	42262	
	*-----.							
	*-----.							
(4)	1	1	0	0	11	54	42102	
	*-----.							
	*-----.							
(6)	0	1	0	0	13	0	0	
	*-----.							
	*-----.							

[[] PAGE BREAK-- end 12-2

LAS - Header for List of Available Space in BLK

LKF
*-----.
42260
*-----.
35 18

LAL - Header for List of Available Space in LCM/ECS

LKF
*-----.
317
*-----.
35 18

[[] PAGE BREAK-- end 12-3

[[]pages 12-4,5,6 -SEE original .PDF FILE PAGES 72 - 74

[[] PAGE BREAK-- end 12-6

ROUTINES

A. Initialization routines

1. IMP • Initialize Mass 1/ Processing

This set of 5 subroutines performs initialization functions for the tables,

flags and linked lists used by the routines that handle dynamic storage allocations.

IMPA - Called from the Bridge initialization to open the random file ZZZZOP and set the flags for LCM/ECS usage

IMPB - called from GPO to dump the last of the phase 1 blocks to disk and initialize the two packing buffers RRB and RWB

IMPC - called from GPO to initialize BLK & LAS for the list of available space for OPT=2 processing. It also sets the OPT=2 table overflow exit.

IMPD - called from GPO to initialize BLK & LAS for OPT*1 processing. Since only one sequence loops are being processed, BLK is set up with the available space at the lower end and the block at the top to reduce garbage collection. The BIT entry for the block is stored here.

IBS - an internal routine called by IMPC & IMPD to do the free space initialization in BLK

B. External Routines

1. WMB/WFB - Write Modified Block to BLK

This routine has two entry points, WMB and WFB, and is used to write a modified block from another table into BLK.

The difference between the two entry points is that WFB sets the final write bit in the block header word & the BST entry.

It is called for blocks that have been coded into save instruction format (SI).

For this entry point the FWA and length of the block are specified as entry parameters, while WMB assumes the block resides in TXT. A call is made to ASB to allocate space for the block in BLK. The actual transfer is then done, the block header & trailer words are stored, and the BST & BIT entries are updated.

There must be an entry for the block in the current BSI.

2. RTB - Read Text Block into BLK

Reads a block specified by the current BSI index to BST from LCM/ECS or disk into BLK.

If the requested block is already in BLK, the exit parameters O.SEQ & L.SEQ are set to the FWA & length of the block respectively.

Otherwise, a call is made to ASB to allocate space for the block in BLK & GMB is called to read the block. The BST & BIT entries are updated and O.SEQ & L.SEQ are set.

[[] PAGE BREAK-- end 12-7

3. RBS - Release Block Space

called to release all space in BLK or LCM/ECS associated with a block specified by the current BSI index to BST & change the BSI & BIT entries to point to a dummy block which consists of a BOS and EOQ

4. RNB - Read Next Block of ZZZZOP File in a Sequential Manner

called from FBV to read all basic blocks for a program either from LCM/ECS/disk. REAONS is used to read the file in 6000 mode.

The routine GMB is called to read blocks from LCM/ECS or for 7000 mode disk files.

5. SMB - Save Memory Block

called to write a block to the random write buffer, LCM/ECS/disk.

The entry parameters give the FWA & length for the block & also the old LCM/ECS address so that the space can be reused. Blocks are transferred directly to LCM/ECS to prevent writing short records to disk, blocks are packed into a random write buffer, RWB, and full buffers written to disk.

Blocks that are longer than RWB are written directly to disk as a single record.

The random index word is formatted and returned as an exit parameter* ASL

is called to do the actual transfers to LCM/ECS and disk*

6. GMB - Get Memory Block - called to read a block from the random read buffer, LCM/ECS/disk The entry parameters give the FWA of the space.
The block is to be moved to and the random index word for the requested block.
Depending upon the address type, the routines GBL or GBO may be called to read the block from LCM/ECS or disk. N.RRB is checked to see if the record containing the block is already in RRB so that only a transfer is necessary
7. DMB - Dump Memory Blocks - called from ASB or the table manager (PROSEQ) to dump blocks from BLK to LCM/ECS/disk when additional space is required.
Blocks are dumped on the basis of priority - first those with the final write flag set, second those not included in the BST for the current region, and finally all others except the block that O.SEQ currently points to.
Blocks are dumped until the amount of free space requested as an entry parameter is obtained or until all possible blocks have been dumped.
A flag denoting success or error is returned on exit,

[[] PAGE BREAK-- end 12-8

C, Internal Routines

ASB - Allocate Space in BLK - Allocates space for reading or writing a block into BLK
The entry parameters give the address of the block header word and old length of the block if a previous copy is already in BLK.
The new length being requested is used to determine whether the block can be rewritten in place. If not, the space occupied by the old copy is added to the list of available space & the list is searched for a larger available block.
If enough available space exists, but not as a contiguous area, the garbage collector GGB is called to compact it. Otherwise DMB is called to free up the necessary space. The FWA of the allocated space is returned as an exit parameter,

ABB - Add Block to BLK List

Adds a block of available space to the linked list of available space headed by LAS.
The FWA and length of the block are given as entry conditions. The blocks immediately physically proceeding and following the block to be added are checked to see if they are also available space. If so, all contiguous available space is combined into one block and made one entry on the linked list,

RBB - Remove Block from BLK List

Removes a block of available space from the linked list of available space headed by LAS.

CGB - Collect Garbage in BLK

Collects the fragmented blocks of available space into one contiguous area.
Entries in BST and BIT are updated for all blocks that are moved. The list of available space is updated to contain only one entry, and MX.AVS, the maximum amount of available space is reset, CGB is called from ASB or the table manager if the size of the table BLK is to be reduced.

There is currently no logic to allow the size of BLK to be increased.

ASL - Allocate Space in LCM

Allocates space for a block in LCM/ECS if it is available.

The entry parameters give the FWA of the old copy of the block in LCM/ECS if one exists and the new length being requested.

The FWA and the length of the block assigned are returned as exit parameters.

The logic is similar to that of ASB except that each block is allocated with a "padding factor" (initially 20 words) to allow a modified block to be rewritten in place more often. When the available .space is greater than 1/4 of the total LCM/ECS field length and no more space is available at 0.LCM, a garbage collection is done,

[[PAGE BREAK-- end 12-9

A8L - Add Block to LCM List

Adds a block to the linked list of available LCM/ECS space headed by LAL.

All contiguous areas are collapsed into a single entry. The logic is similar to AB8.

R8L - Remove Block From LCM List

Removes a block from the linked list of available LCM/ECS space headed by LAL.

The logic is similar to ABL.

CGL - Collect garbage in LCM

compacts all blocks of available LCM/ECS space into a contiguous area beginning at HO\$0FLL.

Entries in BIT are updated for all blocks moved.

This list of available space is updated and MX.AVL is reset.

MVL - Move Block of LCM Data

Uses a 100 word SCM buffer to move a block of LCM data.

Entry parameters are the word count, source FWA and destination FWA.

[[PAGE BREAK-- end 12-10

IOTBLD • Random I/O Table Format Definitions (comdeck).

RI. - Random Index Word

IC Bit 59 =1 in core, =0 on MS

AT Bit 58 address type, =0 disk, =1 LCM

Bit 57 UNUSED

LEN Bits 39-56 block length (incore copy)

OFS Bits 30-38 block offset in record

RA Bits 0-29 disk address of block

FWA Bits 18-35 FWA of block in BLK

LCM Bits 0-17 LCM address of block

The random index word is the first word of the 2 word entry for each block in BIT

BA. - Block Address Information Word

BLK Bit 59 =1 if block is in 3L&

PPI Bit 58 block priority for ouging

HB Bit 57 holding block

FW Bit 56 final write flag

	Bits (5A-55)	unused
BI	8its (36-53)	BIT index (=BN»2)
LEN	Bits (18-35)	block length (Incore copy)
FWA	Bits (0-17)	FWA of block in BLK

The block address info word is the 2ond word of the 2 word entry for each block in BST

BH. - Block Header Information Word

AV	Bit 59	=1 block is available space
PPI	Bit 58	block priority for purging
HB	Bit 57	holding block
FW	Bit 56	final write flag
Bits	(5L-55)	unused
BI	Bits (36-53)	BIT index (=BN»2)
BST	Bits (18-35)	index to BST table
LEN	Bits (0-17)	block length (including header & trailer words)

The block header information word immediately precedes the first word of the block

BT. - Block Trailer Word

AV	Bit 59	*1 block is available space
Bits	(5%-58)	unused
LKB	Bits (36-53)	ABS link backward
LKF	Bits (18-35)	ABS link forward
LEN	Bits (0-17)	block length (including header & trailer words)

The block trailer word immediately follows the last word of the block plus any extra words allocated

[[] PAGE BREAK-- end 12-11

GLOBAL FLAGS

O.LCM	ENTRY. 0	Next available LCM address
MAX,BLK	ENTRY. 0	MAX size Of BLK

LOCAL VARIABLES

MAX.LCM	DATA	200000B	Max LCM compiler uses
O.OISK DATA	1		Next available RA on disk
ADT BSSZ	1		Address type, 0-OISK, 1*LCH
O.RRB BSSZ	1		FWA of random read buffer (RRB)
O.RWB VFD	42/,18/FWAB		FWA of random write buffer
RLEN BSSZ	1		Length of current record in write butfer
N.RRB BSSZ	1		RA off record in RR8
MX.AVS ENTRY.			Max available space in BLK
MX.AVL BSSZ	1		Max available space in LCM
LAS BSSZ	1		Head of BLK list of available space
LAL BSSZ	1		Head of LCM list of available space
MIN.AB EQU	40B		Min size block added to LAS

MIN.AL EQU	100B	Min size block added to LAL
LCM.XL EQU	20B	Extra length added to LCM blocks

[[] PAGE BREAK-- end 12-12

13.0 FBV - FORM BIT VECTORS

FBV forms the use/def & live exit bit vectors used in phase 3 of global optimization. It is called by GPO after the control flow information has been analyzed. On exit from FBV the bit vectors for all reachable program blocks have been formed, and the space occupied by FBV is used as a buffer by MIO for LCM to LCM transfers.

FBV consists of a main loop to read to the program blocks & call FUD in GPO to form the USE, UBD and DEF vectors, & 2 subroutines. SND calculates the node order of the program graph, which is used in CLI to calculate the live on exit bit vectors.

SND is a depth first search of the program graph that assigns a number to a node the last time it visits it, and is straightforward. CLI consists of two loops. First it sets up a loop control vector which is a list of the nodes of the flow graph in SND order. The second loop calculates the live exit bit vectors by repeatedly iterating the equation

$$Lx(n) = V ((Lx(s) \wedge DEF(s)) \vee UBD(s)) \quad [\wedge = \text{and}, \vee = \text{OR}]$$

s is a successor of n.

The Hecht-Ullman paper shows that this algorithm will converge in $\leq K^2$ iterations where K is the number of derived graphs.

To simplify CLI the program entry node is made a successor of the program exit node & the bit vectors are fiddled with in UDT/AUT.

This has the effect of allowing side effects, which in some cases causes the compiler to generate some extra post stores.

Consider the example -

```

SUBROUTINE X
REAL A(10)
00 10 I = 1,N
10 S = S + A(I)
ENO

```

S is live on exit by the equations and it is not really. It should be diagnosed as an error.

[[] PAGE BREAK-- end-13-1

14.0 GPO - GLOBAL PROGRAM OPTIMIZATION

GPO contains the control logic for phases two to four of the OPT=2 global optimizer (code notion and strength reduction) & the control logic for the OPT=1 loop optimization.

The algorithms in GPO are an amalgam of interval analysis, the CSC-LNRA (Linear Nested Region Analysis) algorithms described in Cocke and Schwartz,

and Fran Allen's article in Advances in Programming.

The global optimization consists of a pass over the program, working from the innermost loops out to move out invariant code, reduce integer polynomials (array subscripts) to simple adds, eliminate dead definitions, and assign machine registers across the loops.

After the entire program is processed, the blocks are then processed in their program order, addresses are assigned and the global temporaries are packed.

The subsections of GP0 are:

utility subroutines - WTB, PCC, SRI, SII, SMI, MPS, MTB

OPT-1 loop optimization - COL

OPT=2 initialization and control - GP0, CIC

graph processing routines - IGP, FNL, GNG

region initialization IRP, TRP, IHB, CBT, ATT

block combination - CBB, CHB

use/def and bit vectors - CR3, EBV, FU0

global optimization ROD, IPS, AUV, FXI, SHB, MII, FII, MIP, MIP, CIF, DIF, FIM, EIE, UPB

Overall flow of OPT=2 processing

Phase 1 - (Bridge, PROSEQ, SQZ, UDT)

Process front end output, subdivide program into basic blocks, collect control flow information, eliminate redundant expressions, form use/def variable dictionary, save basic blocks on a random file.

[[] PAGE BREAK-- end-14-1

Phase 2 - (GP0 initialization, UDT, CFA, FBV)

Call AUT to reformat the use/def dictionary, call CFA to analyze the control flow information & form the interval lists. Setup the fixed tables in low mem. Call FBV to read the program text sequentially, & form the USE, UBD, DEF & live exit vectors. Initialize BLK, the in memory page table,

Phase 3 - (GP0, GRA, SQZ, MCG, AIS)

Process the program loops, working from the innermost out to eliminate dead definitions, move out invariant/strength reducible code and assign registers across the loops.

Phase 4 - (GP013-GP036, CGTM/WII)

Linearize the coded blocks so WII will process them in in their source program order.
Process TET and assign CA's to non-equivalenced entries.
Call WII in CGMT to convert the instructions and assign addresses tor CGIA.

Finally, return control to the host so it can do the end processing & call the assembler.

Phase 5 - END Processor, CGIA (Internal Assembler)

Final address definition and assembly.

GP0 Main Loop, Overall Processing (GP01 - GP010)

Call IGP to set up the graph tables and interval list pointer.
If there are no loops in this graph, exit to the straight line code processor (GP011).

Loop processing starts with a call to FNL to find the next interval that contains a loop,
it sets the pointer IBA to the address of the interval list.
IRP initializes processing by allocating and setting up BST, the block status table,
which is used as a "loop control vector" to point to blocks in the SCR (strongly connected
region).

The second word of each BST entry is used to pass information between the various
optimizing routines in GP0 and GRA. It also initializes the flags and tables, etc.

FUD is called to process the blocks & collect the USE/OEF information in UDT.
If there are any user function calls, then EBV is called to set the region definition bits
for common variables ..., so as to inhibit optimization of any expressions involving them,

RDD calls IPS to insert any post stores that were created during inner loop register
assignment. Then it scans the block for dead definitions, and eliminates them.

[[PAGE BREAK-- end-14-2

MII processes each block to mark the invariant instructions, & add them to the block IIC.
It also forms STC, the chain of potential recursive definitions.
MII also forms the region use and def bit vectors.

IHB initializes the holding block, moving the loop label out if necessary.

FXI forms the region exit information for GRA, which includes the region live entry bit
vector, the movable definitions, and the live entry bit vectors for the exit nodes.

Global optimization continues with FII, which scans the store chains tor recursive
definitions and forms a list of their increment values in IIT. If there are any then
HIP is called to process each block in the region to mark the integer polynomials,
a linear function of a recursively defined variable; and to add them to the IIC.
It also collects information used for test replacement.

MIE scans all the blocks to move out invariant & strength reducible expressions that are
profitable to the holding block & to mark the places in the blocks where temporary loads
should be inserted.

SHB is called to squeeze the holding block and eliminate redundant temporaries, which is
done by calling SQZ.

If any terminal polynomials were moved out then CIF, DIF, FIM and SHB are called, CIF collects the IP and increment formulas associated with a terminal integer polynomial (TP) and saves them in IIT. FIM rescans STC, and for each recursive definition it builds a list, of the TP's that have to be incremented at the store and forms the increment value by calling EIE. SHB is called if any variable increment code has to be added to the holding block.

UPB processes all the blocks to replace the moved expressions with temp loads (TLD's) & insert the compensating increment instructions after recursive definitions, and rewrite the modified blocks. It is the final phase of the machine independent optimization.

C8B combines basic blocks in the region into extended basic blocks and calls RIO to - Jam order- the EBB's. It also expands any level 2 references.

GRA is called to do global register assignment & code the sequence.

Finally, TRP is called to terminate processing of the region.

Blocks outside of all loops are processed by CIC, who assembles an EBB and calls CHB, RDD, CBB and CXB to code the blocks.

Phase 4 scans BIT to order the blocks in source program order, It calls assign address and writes them to SLIST.

[[] PAGE BREAK-- end-14-3

Subroutines (listing order)

AFB Allocate storage for a fixed table in low memory, GPO initialization only

STO Set managed Table Origin (low memory limit), GPO initialization only

WTB Write Text Block in TXT back to BLK after it has been modified.
Exits with L.TXT=4 (BOS only)

PCC Process Coroutine Call - called by the CLCM macro to call the routine that is its entry argument for each block in the region (in BST) except the holding block.
It calls RTB to make the block accessible (put it in BLK), sets up O.SEQ, L.SEQ, BN, BSI & the registers & calls the routine. It updates BSW & BST which the called routine may have modified.

SRI/SII Store an IL instruction in a table, B6, B7 » table index & FWA. B2, X6, X7 = opcode, RI, R2 words.
SII sets the INC bit in the descriptor word of the instruction.

SMI Same as SRI, but also uses a memory reference and sets FP/LZ bits in descriptor.

COL Code Optimizable Loop (OPT=1), a quick and dirty trip through GPO and GRA.

GPO Initialization, main loop, phase 4 of global optimization.

IGP Initialize Graph Processing. Setup graph table pointers used in GPO, initialize BTT & HNT.

FNL Find Next Loop. Scan interval lists for next one with a SCR (loop)●

GNG Advance graph table pointers to Next Graph, or read it from disk.

IRP Initialize Region Processing. Set up BST for a region.
Setup HB in BIT, initialize flags in /GPOGRA/, set LGL if multi-pred IF loop.

IHB Initialize the holding block (HB) in TXT.

TRP Terminate Region Processing for a loop. Final processing of HB,
reset UDT, clear tables, collapse BST.

CIC Convert Interval to Code. Meant to be used primarily for straight line code.
Accumulate EBB's, do some optimization and code them.

ATT Adjust Temporary Table TET to clear REG field & update IPI fields of equivalenced entries.

[[] PAGE BREAK-- end-14-4

CBB Combine Basic Blocks in an interval to EBB. Expand level 2/F.P. refs,
merge the blocks, re-squeeze and Jam order.

CRB Clear Region Bits UDT. Resets UDT after processing a region.

EBV Expand Bit Vector to set bits in UDT.
Given a set of elements in UDT. Expressed as a bit vector & bits to be set in UDT for
elements in the set, EBV scans the bit vector and sets the bits.

FUD Form Use/Definition information and bit vectors.
FUD scans a basic block (with IO list info) for memory references and jumps and
sets usage bits in UDT and collects a list of marked variables. The list of marked
variables is used to save time in resetting UDT, and in forming the bit vectors.
Bit vectors are formed if the flag BBV = 0. When FUD encounters a HB it is processed
specially, and CH8 is called to see if it can be combined with its predecessor.

CHB Combine Holding Block with immediate predecessor. CHB is called fairly early, so that
code moved out of inner loops is subject to code motion out of outer loops. It checks
to see if the block it is processing is a HB to an empty block, and adjusts the bit
vectors of both blocks so that further optimization can take place.

RDD Remove Dead Definitions from a block. Call IPS to insert post stores and adjust DEF,
UBD bit vectors. check the block bit vectors for stores that are dead on exit and not
used in the block, First i.e. $X = A \text{ } \$ \text{ } Y = \text{SIN}(B) \text{ } \$ \text{ } C = X$. Y & C may be dead
If there are any potential dead definitions,
then scan the block backwards for dead stores, changing any to NOP's,
and adjust the PS bit of store predecessors.
Finally if dead stores or dead computations were found, re-squeeze the block to remove them.

IPS Insert Post Stores that were created in GRA into a block.
During register assignment GRA may move the definition of a variable from the loop body
to its exit nodes. The moved definition is called a post store.
The list of post stores for a node is kept in PSI and the PII field in PIT points to it.
IPS checks the PII field of the block and if zero, exits. In the case that a post store list
is present, the block DEF and UBD bit vectors are adjusted, the store instructions are setup
and inserted at the beginning of the block. Finally AUV is called to adjust the USE vector.

AUV Adjust USE bit Vector of a block after a SQZB call to account for any store/load squeezing that took place. Rescan the block and reform the USE vector in SVA. Exit if block contains any user or I/O external references. Otherwise, set block use vector to SVA, and the UBO vector to SVA and UBD.

[[PAGE BREAK-- end-14-5

FXI Form loop entry/eXit Information for use by GRA.
Compute region live entry bit vector (LEA). Form list of exit nodes of the SCR.
Scan the exit list and form the list of exit nodes, their live entry bit vector and if the exit node can be post stored into or not in RXI. Also form the moveable def bit vector which the set of definitions that can be moved out of the loop.
FXI forms the region exit information in RXI which is printed out be PRNTRXI.

SHB Squeeze Holding Block. Terminates the HB with an EOQ and calls SQZB to eliminate redundant instructions and redundant temp stores. Rewrites the HB back to BLK.

MII Mark Invariant Instructions and collect information for strength reduction.
Forward scan of a block to form a list of invariant instructions in the link words, mark stores as not recursively defined in UDT, collect a list of possible recursive definitions, substitute label references (multi-pred IF loops), count number of references to the loop header.
When EOQ is encountered, save the invariant and store chain pointers in the link word of the BOS and the block status word. Accumulate the SCR USE and OEF bit vectors.

FII Form Increment Information. Process blocks with STC's.
Scan the STC to eliminate non-recursive definitions, and save the increment values of the RD's in IIT. Point the store instructions to them. Mark the ST and its predecessor as INC in the descriptor words for GRA.

MIP Mark Integer Polynomials. An integer polynomial is a linear function a single recursively defined integer variable. MIP consists of a forward scan of the block to mark the IP's, compute their cost and form a list of them in RND. When a JPX is encountered, we check for a reference to the header node, & if so collect information for test replacement in GRA. When EOQ is encountered the IP list is merged with the IIC chain in the link words.

MTE Move Invariant Expressions and terminal integer polynomials to the holding block.
First the IIC is scanned for terminal invariant or strength reducible expressions that are profitable to move out, and a forward chain of them is formed in the link words.
Next space is allocated in the HB (in TXT) and the instructions are moved out and setup as IT.(K) = EXPR in the holding block.
The link word of the temp stores are chained, and they point to the place in SEQ where a TLD of the moved instruction should be placed. The operand R-numbers of the instructions added to TXT are adjusted to bring them to canonical form. The chain of moved instructions in SEQ is rescanned for stores and RJ's and they are changed to BOS's so SQZ will eliminate them when the block is updated. Finally, the information about the insertion points of the temp toads is saved in TET for use by SQZ, UPB, etc.

[[PAGE BREAK-- end-14-6

CIF Collect Increment Formulas. CIF collects information about terminal polynomials in IIT that is needed later in GPO to develop the increment of the IP when the independent variable is incremented. and in GRA for the test replacement, counting method decisions.

CIF develops the information in the B list of IIT (see GPOCOM and CIF snaps), for each terminal polynomial in the HB. First a special call is made to MII to mark the invariant instructions in the HB. The HB is scanned a chain of the TST's that define TP's is formed. The chain is scanned and the increment and polynomial formulas are developed in IIT by calling DIF. The ITI field of the TET entry for the TST points to the IP information in IIT. Finally, the IP lists in IIT is scanned for similar IP's! those with the same increment formulas; and the IP's are marked as similar, and the first is marked as a base member.

DIF Develop Increment Formula of an IP of a single variable or extract an IP formula. DIF is called by CIF to extract the IP formula or increment formula of an IP in TXT and place it in IIT.
To this end it uses RND as a scratch table. The algorithm consists of two phases. A top down parse (to extract the formula from TXT and form an output list in RND, and an output phase to expand and move the instructions from RND to IIT. In order to form the increment formulas of an IP we note that

$$\Delta IP = IP(J+\Delta J) - IP(J) = \text{DERIVATIVE IP} * \Delta J \quad \text{where } \Delta \text{ is the Delta symbol}$$

FIM Form Increment Modification lists. Process blocks with STC's to form a list of the TP increments at each recursive definition for use by UPB. At each recursive definition on the store chain the IP list in IIT is scanned for IP's that depend on the variable being defined.
The increment formula is evaluated at this point by a call to EIE and the increment value is saved in IIT. The ST is updated to point to the list of increment values in IIT.

EIE Evaluate Increment Expression. Evaluates del IP (ΔI).
Attempts to evaluate the expression itself or calls SIE in SOZ and adds the expression to the holding block.

UPB Update Program Blocks in the loop to reflect effects of code motion and strength reduction. UPB modifies the blocks by inserting TLD's of the moved expressions, IP's & adds compensating increment instructions to the blocks by setting up instructions in MOD, and mod insertion list, & calling MPB to merge the mods in, and re squeeze the block. First TET is scanned and inserts are set up for TLD's of the moved expressions. Next STC is scanned & the compensating increment instructions are formed. Finally the block is updated.

[[] PAGE BREAK-- end-14-7

MPB Merge Program Block (in BLK) with TXT and mods to block in MOD, MLT.
MPB is a general subroutine to merge two blocks (concatenate), and/or a modify a block. On entry TXT may contain sequence of instructions (BOS ... EOQ) or Just a BOS. The pointers O.SEQ, L.SEQ define a block in BLK that is to be modified according to the insert/delete info in MLT and the instructions in M00 and appended to TXT, after the last instruction in TXT is removed. On exit, the result is in TXT, or written back to BLK, with the R-numbers adjusted, the resultant block squeezed, etc. according to the value of the entry flag.

The algorithm is -

Allocate space in TXT.
Scan M00 list and mark instructions in SEQ that are to be modified and
save MLT info in the link words of the instructions.
Move the instructions from SEQ to TXT, and insert the new instructions from M00.

Scan TXT and adjust the operand R-numbers of all the
instructions to bring them back to canonical form.
Call SQZB if requested, rewrite block back to BLK if requested.

MT9 Move Text Block from SEQ to TXT and clears the link words in the process.

[[] PAGE BREAK-- end-14-8

15-0 GRA - GLOBAL REGISTER ASSIGNMENT

GRA is the machine dependent loop optimizer.
Called by GPO after it has performed any possible machine independent global optimizations.

Optimizations performed by GRA are -

The motion of loads and stores of scalars from a loop to its entry/exit paths
and the assignment of the quantity to a register in the loop.

Prefetching of indexed loads in small innermost loops to reduce the critical path time.

Code size reduction in the loop body by assigning SCM address's & constants to the registers.

Reduction of IA to SA instructions.

Elimination of useless variables.

Linear function test replacement.

During the development and testing of the compiler, GRA was the hardest to write,
and the buggiest of the routines.

Most of the problems stem from the non-homogeneous nature of the machine.

Algorithm Overview

MARA, the maximum number of A-assignments is initialized to 4.
The blocks in the loop body are scanned for register candidates & their usage mode.
The usage of a candidate determines what type of register it will go in,
and if it can be subsumed (have an address appended).
The candidate table is scanned & entry/exit information collected by FXI is filled in.
The register assignments are then selected from the candidate table in the order -
variables that are B-candidates, X-candidates, prefetch candidates.
The candidate table is rescanned, and address candidates are picked.
Finally small constants are assigned to the B-regs & the final loop counting method
decision is made.
The blocks in the loop body are modified and MCG is called to code them.
The initialization code is setup in the HB and the post store lists
& other exit conditions are setup.

The basic algorithm used is local register allocation (REG width),
1 to 1 global assignment, followed by local assignment.
The algorithm is a simpler version of the one described by Beatty.

Structures DELIMITER

- RAT Register Assignment Table.
24 word table holds the initialization formula for the registers
- RCT Register Candidate Table. Variable length in managed table area, 3 words per entry.
RCT holds information about the candidates & acts as a global expression dictionary.
First entry is zero, a search terminator. Second entry is the constant 1.

Link words are used to point to RCT, and during IRA they hold the R-numbers of the different registers that a value may be in.

The INC bit of ST's & IA, IS, STT instructions are set if they are increments.

/GPOGRA/ holds information about the loop that was collected by GP0.

The blocks are -JAM- ordered, and the max reg width of the loop is in MAXW.

Routines (functional order)

- GRA control, calls all the major modules
- IRP initialize processing. Set up RCT, zero scratch area.

Candidate Table Formation Routines

- DUM Determine Usage Mode and candidate type of instructions.
Backwards scan of the blocks to determine how the result of an instruction is used.
The possibilities are:
- EU - explicit usage - cannot be biased by an address, may have X-uses in the sequence.
RF - short usage - Lower 18 bits used as RF of an indexed LD/ST.
IA - used as operand of IA or IS. Conditional EU.
TU - Test Usage - used as part of a loop back test when loop is DO loop or
 recognized as one.

The information collected in DUM is used to decide what type of register (X or B)
a candidate may be put in. The usage mode bits are kept in the link words during DUM.

- ERC Enter Register Candidates. A forward scan of the blocks to enter candidates in RCT &
extend assignments that were made in inner loops. When a possible candidate is
encountered (LD,ST,S,FMA ...) SCT is called to enter the candidate and setup the
link word which points to the RCT entry.

Link words of non-candidates are cleared. When a full lock RS is encountered, its
predecessor is checked to see if it is in RCT, and the RS, pred are checked to see if
a register assignment made in an inner loop can be extended to the loop being processed.

[[] PAGE BREAK-- end-15-2

- SCT Search Candidate Table The search & enter routine for RCT that is called by ERC.
It searches RCT for a match, & if none, makes a new entry. The link word of the candidate
is set to the RCT ordinal. When an increment store ($I = I + INV$) is entered, SCT collects
information about the increment value, number of times the variable is incremented, etc.
- SEE Set Entry/Exit information for the value candidate.
SEE is called after ERC has entered all the candidates in RCT. It scans RCT and fills
in the entry/exit bits that were setup by FXI in the region bit vectors.
It also checks for candidates that are defined in the loop, dead on exit,
and have no real uses in the loop. These are marked as RA and KD, so that they are killed.
Finally RAT is scanned for moved assignments (ERC/RS) and the INV bit is set if necessary.

Candidate Selection

- MTA Make tentative B-register assignments, and select a method to count the loop (DO loops).
Setup a sort table of value candidates that are RF (short usage) and RD, and sort it on
the sort key fields (TU, PRFT, SUSE ...) If the loop is a DO loop, then the -first- entries
in the sorted table will be the loop index, limit and increment.
Assign them to B-regs first, and based on the setting of the bits (LX, SUSE ...).
Determine a possible counting method. Next the rest of the candidates in the sort table
are assigned to the remaining B-registers. Finally if no test replacement will be done
then the EU bit is set for candidates that are TU, & games are played with the NOCC
field of the increment values to force them to registers if they are constants.
- AIR Assign Index Registers.
Assigns a candidate to the next available index register if it has not already been assigned.

[[] PAGE BREAK-- end-15-3

- DXA Determine X-Assignments.
DXA selects X-register candidates for an innermost loop, and validates candidate selections
made in an inner loop. First an X candidate table is formed and sorted on the sort fields
(PRFT, SUSE, etc.).
The MIN (4, X-candidates) are selected and the RA bit set for them.
The X-candidates are regarded as tentative.
DXA now calls CRW to see if the max reg-width with the B and X-assignments is less than 9.
If it isn't then the number of candidates is decreased and the above repeated.
The result of this process is that we have assigned the max number of X-candidates in the
loop for a sequential machine. Next CMR is called to count the number and type of memory
references remaining in the loop. Finally XCT is scanned, and registers are assigned to
the candidates based on their usage in the Loop.
- CRW Compute Register Width Computes the max X-register width of a block with the assignments
made in MTA & OXA. The algorithm is similar to CRW in BDT, except that the register width
is not changed for candidates that are locked in registers.
- SLW Save Link Words (temporarily) in RND) in case they are needed again.
- RLW Restore Link Words.

CMR Count remaining Memory References in a block.
 Scans block to count the number of memory references that will remain after register assignment.
 This information is used by DXA to determine X-assignments, & DAA to determine the max number of A-regs that can be allocated.
 NOTE: if any block in the loop contains a store it would not be wise to tie up both X6 & X7

DAA Determine A-register Assignments. After B and X assignment have been made, the code in small one or two sequence loops can be speeded up substantially by prefetching indexed loads that are on the critical path, and overlapping the fetch for the next iteration with the loop back Jump. Because of the excessive amount of code in the compiler required to produce optimal code the logic in DAA is rather simple, & produces suboptimal code in some cases.

DAA first checks to see that the loop is -small- & based on MARA and the counts developed by CMR computes MAA, the max number of A-registers that it will assign. Next it reads and scans instructions in the header node (skipping the BOS and LAB); until it hits a boundary marker for indexed loads whose index is not constant and RD.
 The class the LD is from may not be defined in the region. Information about the candidates is saved in ACT, the A-candidate table. At this point we should run a critical path calculation, & sort the A-candidates on their "slack", but we don't.
 The order that the candidates in ACT are in is the same as their order in the sequence, which is JAM order, which is an approximation to critical path order.

The candidate table is now scanned to remove candidates with large constant or variable increments (UO not selected). (UO = Unsafe Optimization - Array ref may exceed FL)
 Finally ACT is scanned X-registers are assigned, RCT & RAT entries are updated, and B-registers are deallocated if necessary.

i[[PAGE BREAK-- end-15-4

MFA Make Final B-Assignments. Up until this point candidate selection has centered about VC's (value candidates), whose affect, when assigned to registers in the loop is quite profitable. MFA concerns itself with reducing the size of the loop by assigning constants & addresses to B-registers, appending addresses to value candidates that have no explicit uses, and reducing the number of increments by address differencing. As far as the design goes it is somewhat weaker than the previous DO processor, mainly because it doesn't catch some common cases, but it is fairly difficult to do a very good Job here.

The algorithm is as follows -

Scan RCT and form a list of constant candidates in RCT and a sort table of (8C) of unassigned address candidates in RND, Sort BCT on RF & IH and build an index table for each RF class and sort it on if the RF class is BASE (base member of a similar class or not), etc.

Starting at MFA use the RF index table as control, append addresses to the assigned value candidates, form constant differences and enter new con candidates, do address differencing ... Until we run out of B-registers.

Finally scan the constant list, form a sort table and assign constants. Then go back & modify any instructions of the form PLD RF+CA that might be affected by the constant assignments. Finally check the loop test replacement decision, adjust it, and set the necessary flags for IRA, SUP.

The RF classes type processed by MFA are -

1 RF, IH in class.

There are two possible cases depending on whether the RF is BASE or not.

Consider a loop on I with the references U(I), U(I+N).

The first ref would be the base, and second could be written as the first +N.

During GPO these are separated into 2 separate integer functions, with their own increments. MFA eliminates the second increment by assigning the difference $I - (I+N)$ to a register instead of $I+N$.

1 RF, IH, 2 or more DA's

●●●SIJ NOTE I don't like this example OR the solution

Example $A(I-1) + A(I) + A(I + 1)$

call PSC to find the center of the class, and change the refs to $A(I-i*C)$, etc.

Where C is the center. Enter constant candidates for the difference entries, and mark them to be setup as PLD/PST's.

[[] PAGE BREAK-- end-15-5

More than one IH

Try address differencing if there are enough B-registers to go around.

Partial assignment case tries to put $IH+CA$ in a register

if subsumption is impossible or there is no RF.

PSC Process Simple Class. Process a class with 1RF,IH and many CA's.

AAD Assign Address Difference to a register - Change a ref to $CAIH,RF$ where RF, CAIH, is in a register to express it as an address difference. Set up a second register as $CAIH - CAIH(2)$ or the reverse.

EDC Enter a difference candidate in RCT.

SDC Search B-registers for previous occurrence of a difference candidate.

AFA Assign full address to a B-register. Change a B-register assignment from the RF to assign $CAIH+RF$ to the register.

ECC Enter a Constant Candidate. Search con list and enter a new constant or bump numbers of occurrences of it (used as sort key in determining assignments).

CLB Code Loop Body (control). Because of the various difficulties MCG can get into, the register assignments made in DXA must be regarded as tentative until we have successfully coded the loop body. The basic function of CLB is to modify the loop body to reflect the assignment mode in the candidate selection phase and call MCG to code it. If X-assignments were made, the coded blocks are saved in PIT until all the blocks in the loop body have been coded. Then they are written back to BLK. When MCG cannot code a block in the loop because there are too many X-assignments, the last one is removed, the tables adjusted, and CLB tries again. If the failure is due to too many A-Assignments then MARA is set to NAA-, the tables cleared and CLB exits to the main loop (GRA0).

The main loop in CLB is -

Initialize flags, form post store list.

For each nonempty block call IRA to modify the sequence (insert the register assignments).

Call CUC to set the uses counts and delete any useless instructions inserted by IRA.

Call CIS to set the initial parcel count and code the sequence.

Collect the set of registers used in the loop.

Save the code length info in BST and advance to the next block.

On exit from the loop (success) the coded blocks are transferred from PIT to BLK.

[[PAGE BREAK-- end-15-6

CXB Code Extended Block. Code a block that is outside of all loops.
 AIS is called to assign B-registers in the sequence, if profitable,
 CIS to code the blocks, and WSC to write the coded block back to BLK.

CIS Code Instruction Sequence.
 Check BST, BIT to set initial parcel count and call MCG to code the block.

WSC Write Saved Code from PIT to BLK.
 Rewrites a block that has been coded back to BLK, and sets/updates flags in BIT.

IRA Insert Register Assignments. IRA is functionally similar to UPB in GP0.
 It modifies the sequence to reflect the optimizations performed and places a copy of
 the optimized sequence in TXT. Because of the variety and complexity of the optimizations
 performed in GRA it was decided that it would be simpler to do a copy/modify on the fly,
 then to set up mods and call MPB. Other factors that influenced the decision were that the
 number of mods is approximately the same as the original sequence, and that the
 IRA approach was more flexible.

The basic algorithm is to allocate sufficient space in TXT for the modified block.
Copy the R1 word of the BOS, any initial label, and setup initial full lock DEF's & DAR's
which are the sequence initial conditions.

Now scan the rest of the sequence from begin to end. If the instruction is not a candidate,
then copy it to TXT. If it is, then modify it, etc. During the copy the R-numbers in TXT
setup in canonical order, referencing the proper registers by keeping track of the
R-numbers are that a result may have in different register types in the link words of the
instruction in BLK.

An instruction may have up to 3 R-numbers associated with it. The R-number or value that
is in an X-register, a B-register, & a store register.

For example, suppose a candidate is assigned to a B-register, but one of the instructions
that uses it needs its operands in an X-register. When we process a LD or ST of the
candidate we setup the basic R-number as the in B value, output a SA instruction, & set
its result as the X-value in the LO or ST predecessor instruction in BLK. In this way
useless instructions may be introduced,
a backward pass by CUC is sufficient to detect & eliminate them without involving SQZ.

[[PAGE BREAK-- end-15-7

IRA contains a special processor for most of the instructions, some of them are -

EOQ - output any loop epilogue post stores.

IA,IS - reduce to SA,SS if possible.

XMT - delete if possible.

RS - delete if used is a candidate that was assigned to register (IRA).

DEF - skip DEF if outer loop and IRA

JPX - output prefetches for next iteration, convert to JPBB if test replacement.

LD,TLD - change to address load or PVC

STT - change to SA, SS, etc.

ST,TST - change to short store or process redefinition of a register the Bridge (candidate),
output ST if it cannot be moved.

PVC - initial definition of value candidate encountered,
output SA if necessary, setup link word.

PAC - Process address candidate to change a LD,ST,STT to an appropriate 15 bit instruction.

APD Adjust previous definition to make old value unavailable if a use extends past a redefinition

CUC collect uses counts. Backwards scan of a sequence to collect uses counts and mark useless instructions. If no useless instructions then exit, else compress and renumber the sequence.

ENTRY CODE

SUP Set Up Preloads of locked registers in the holding block.
SUP controls the output of the loop initialization code.
The strategy is setup the initialization code in MOD,
inserting it at the end of the holding block,& call SQZ to eliminate the redundant code.
For all this to work smoothly, without a large amount of special code in SQZ a new IL
instruction, the ILD had to be added.
The ILD is a type III load, that has the property that when a ST/ILD combination
is encountered, not only is the ILD eliminated as redundant, but the ST is also eliminated.
In this way we remove TST's of IT,'s that have been assigned to registers, and later when
the HB is merged with its natural predecessor in GPO/CHB we kill the ST's of variables
that are initialized in the prologue without having to use RDD.
The code in SUP does the following -
insert the values of UXR and MRA in the R2 of the HB BOS for use
by outer loops. Get space in M00 (overestimate) to setup the initialization code.
If TRD=4 (F(i), F(n) in the registers), then setup the upper limit register as F(N) by
looking up the polynomial formula in IIT. substituting N and moving the formula to MOD.
Scan RAT and setup code to initialize the 3-registers.
Scan RAT and setup preloads of the X-registers. Merge the initialization code with the HB,
squeeze out redundant expressions, and write it back to BLK .
Finally, clear the HBN field of IT's created in this loop, (IT'S ???)
scan the HB and set the HBN field of it's defined in the HB,

then scan TET backwards and remove trailing entries with a zero HBN field.

[[] PAGE BREAK-- end-15-8

LIV Load Initial Value. Output S/FHA/CLR or LD of initial value. Calls OIL and SIV

OIL Output Initial LD.
Output LD or ILD of a candidate depending on IF it is dead on exit from the loop or not

SIV Set Initial Value. Generate STT, etc. To form initial value of an address candidate.

OSI Output STT Instruction. Given CAIH and H2 it generates an STT instruction.

ORS Output full lock Register Store (RS) instruction to initialize a register.
Sets INV bit in S0 field if set in RAT.

SXC Set exit Conditions. SXC presently has two functions -

a. To setup post store lists for variables that were defined in the loop
& are live on exit from it.

B. To accumulate LUV, the Loop Usage bit Vector, of variables that were referenced
in the loop, so as to inhibit assignment of them to registers in an outer Loop.

The post store mechanism is described in GPO/IPS.

SXC scans RXI and setups the post store lists in PSI for the variables which were
defined in the loop and are live on entry to an exit block.

In the case of a nest of loops that have the same exit node,
and variables are live on exit from both the inner and outer loops,
SXC merges the previous list with the one being built.

To accumulate LUV we scan RCT and set bits in it for variables whose values were
not assigned to a register in the loop. We also clear bits in LEA,
the loop live entry bit vector for variables that are KD, useless increments.

[[] PAGE BREAK-- end-15-9

16.0 PROSEQ - SEQUENCE PROCESSING CONTROL

PROSEQ contains the sequence processing control logic for pass 2 and many utility subroutines.

CG\$PAS is the sequence processing controller
that is called by when a sequence of instructions is accumulated.
It terminates the accumulated sequence with an EQ,
and depending on the OPT mode calls the various code generation routines.

In OPT=0 or 1, it calls SQZ, PSB & MCG. In OPT=2, it calls PBB.
After processing the sequence it resets the flags for the next sequence and exits.

PSB Propagate Symbol Bits
propagates bits from symbol table to descriptor words of memory references for later use
by other processors. The idea was to reduce the number of references to the symbol table
in the -language independent- parts of the code generation phase.
Presently, the formal parameter (FP) and level bits are placed in the descriptor.

AIS Assign Index registers in Straight Line Sequences
This routine accepts a non-loop sequence of IL instructions in table SEQ,
& exits leaving a transformed copy in table TXT• The transformation consists of B-register
utilization for SCM address values in portions of the sequence which are liable to -Jam- at
scheduling time. We ascertain this by examining the register width of the sequence at each
instruction, placed in the IL descriptor words prior to AIS entry.
This offers some (not Perfect) forecasting of the number of pseudo X-registers which MCG
could use when issuing the sequence. If this width exceeds actual resources extra effort
to place -short- intermediate values in B-registers will probably be profitable.
In fact, the number of registers required by the sequence must exceed a machine-dependent
value, TRW, before B-register assignment will be attempted.

The sequence as passed to AIS may actually consist of several basic blocks.
The boundaries between basic blocks must first be determined,
as B-register assignments and uses are made on a basic block basis.

AIS is composed of two passes over the sequence.
The first pass (or prepass) is subroutine LBM - locate boundary markers.
It makes a backwards scan over SEQ chaining basic-block demarcating instructions through
the link words. In this chain (elements of which are described by the BM. DESCRIBE/DEFINE
structure) is also installed the maximum register width encountered in the following basic
block and the number of -RF- or short values defined in the basic block.
These RF definitions are our B-register candidates,
and have been previously marked RF by SQZ because they appear as address modifiers.

[[PAGE BREAK-- end-16-1

Following the LBM pass the various tables that are required are allocated,
based on various counts made in the prepass. These tables are reformed for each basic
block during the final pass of AIS.

BCT B-register candidate table. The BCT contains information regarding each RF candidate's
profitability within the subsequence (basic block).

MLT & MOO contain the actual instruction modifications after assignment.
These tables are described in MPB (in GPO). They are cumulative throughout AIS pass 2,
that is they are not restarted for each subsequence.
MPB is called once before AIS exits to modify the sequence.

DUT Oef Use Table.
A primitive "tree" of the basic block which is used to locate use spans for the candidates.

PBT Preassigned B-register Table. Contains B-register uses that are incoming with the sequence
in the form of DEF or RS instructions. Some are temporary, as indicated by a TXT index in
the LU (last use) field of the PBT entry, while others are full-locks for the entire
sequence, as indicated by a -infinite" LU.

In addition to the dynamic tables, two 7-word tables are used:

CAT Current Assignment Table - marks a B-register as assigned in this basic block.

PBI Preassigned B-register Index Table.
This table contains pointers for the B-registers to the next pertinent PBT entry.

It also has an "NA" bit which marks that register permanently unavailable (full-Locked).

The second (actual assignment pass) consists of a main loop, iterated once for each basic block.

If there are RF candidates in the subsequence ($BM.NPF \neq 0$) & the maximum subsequences register width is greater than TRW, we proceed with the following phases:

BDU Build the DEF USE tree - DUT and insert the range of entries for each R-number definition in the defining instruction's link word. BDU also forms the basic BCT with RF definitions encountered and the PBT with RS & DEF to B-register encountered. BDU is the final examination of the -source- sequence & represents pass 2 proper of AIS.

IXU Install Extended Use Spans.
IXU completes the BCT by installing the extended uses (span of uses of all uses) for each entry as well as the maximum register widths over the extended use span (a prime profitability criterion).

If IXU indicates that at least one candidate exists with a large enough ($.GT. TRW$) maximum register width over its extended use span we continue with IUP

[[] PAGE BREAK-- end-16-2

IUP Insert Use info into PBT.
This routine completes the PBT by including the use span for each preassigned B-register value, indicating the length of time that each register is unusable in this subsequence due to incoming Locks.

Finally,
ABR Assign B-register is called to assign as many BCT entries as possible to actual B-registers, Short uses of the BCT definitions are modified to use the B-register copy. We prevent reuse of that B-register until uses of the uses of the register have all gone to zero.
This ensures that the Jam scheduler will not have to store out the chosen value for later use (as long as the sequence is not reordered by the scheduler).
While ABR makes no sophisticated attempt at achieving an optimal packing of the candidates into the available B-registers, it does carefully allocate and deallocate the registers taking advantage of any -holes- left by incoming assignments. Thus, if BI is used by unpack/left shift combinations, it may nevertheless be assigned as an RF intermediate by AIS between such temporary uses.

Similarly, many assignments of one B-register may be made in the same basic block, as long as they are for non-overlapping RF def-use extended use vicinities.
It is clear that any reordering of RF defs or their uses subsequent to AIS may cause deadlock situations in MCG.

After the BDU-ABR phases have been executed for each basic block, MPB is called to merge the modifications in MOO into the sequence under the direction of MLT. Finally, CUC is called to reset the USES field in the instruction descriptors.

ESR - Expand Special References

ESR Expand -Special- memory References on a sequence-by-sequence basis.
Depending on OPT level, loop nesting, etc.
ESR is instructed to expand are of three sets of reference classes:

- a. LCM (level 2 or 3) references only
- b. LCM references and all formal parameter references
- c. LCM references and f.p. address references only

Expansion mode is determined by X5 on ESR entry.

Address references are -set X- (STT) instructions,
while 'reference' implies address or memory (load/store) instructions.

The philosophy of LCM and f.p. processing is one of isolation of special casing.

Pass 1 does not differentiate between f.p. or LCM & other symbols except for syntax checking

[[] PAGE BREAK-- end-16-3

All expansions including references of A0 for f.p. addresses & SCM pointer cells for LCM=I mode level 2 references are performed by ESR.

ESR guarantees bit 59 is preserved for LCM addresses
as well as 24-bit integrity for LCM=I mode references.

The first pass of ESR is MSR - Mark Special Reference.

It is a backwards scan over the input sequence (in table SEQ) which chains all special refs to be expanded through the instruction link words.

The second pass is GSR - Generate Special References.

GSR loops through the marked instructions generating sequence modifications to table M00.
These modifications include all necessary addresses loads, 60-bit adds, sets and LCM-access instructions to properly reference the special symbol.

Extraneous instructions are avoided for adds of zero, etc.

If 18-bit arithmetic has been performed on an LCM address (for a LOCF or store to APLIST), instructions are generated to OR in bit 59.

The task of generating special references is divided into several including:

- ISC Issue Set Code performs f.p. and LCM address reference expansion
- ISI Issues Set Instruction IAL - Issues f.p. address loads
- IRA Adds in variable index to address calculators
- ISX Issue Set or transmit following address load
- IPR Issue PLD or PST formed parameter reference instructions
- IDR Issue Direct LCM References
- IMOE Issue Mask and bit 59
- TRE Terminate Reference Expansion files mods to M00 table for a single input reference

After all special references have been processed,
MPB is called to merge M00 with sequence and ESR is exited.

[[] PAGE BREAK-- end-16-4

17.0 MCG - MACHINE CODE GENERATOR

MCG is responsible for scheduling of a sequence of instructions and the assignment of registers (locally), which is usually shortened to scheduling & local register assignment. The main reason for its existence are the multiple functional (arithmetic) units on the 6600 & 7600 machines which enable parallel execution of instructions.

The input is an IL instruction sequence in TXT with the various fields in the descriptor words set (SQZ output).

The output is the posted instruction sequence in PIT in "SI." Format where the first word of PIT is a BOS and the last is a NOP,

Constraints - when not JAM mode the RI, R2 and descriptor words are read only as no copy of the instruction sequence is kept,

MCG uses the PERT-Time scheduling to order the instructions along with a simulation of the machine resources (registers & functional units) to minimize execution time.

In PERT terminology (Program Evaluation & Review Technique) the instructions are activities to be scheduled, and the edges of the network are the operand dependencies & logical constraints between the instructions.

BDT (Build Dependency Tree) forms the PERT network for MCG in TREE & computes the late start times, etc. MCG then orders the instructions by a topological sort that uses the earliest start time as a sort key in addition to the late start time (LST) that is inherent in the order of the instructions on the issue candidate list.

Data Structures

The TREE or USES index table is a list of the successors of a node (instruction) sorted by predecessor and late start time. Examples of the successor index table, etc. may be obtained by assembling BDT with snaps on.

The link word of an instruction holds the *IN0EX* to the successor list, the late start time of the instruction (LST), the number of remaining uses of an instruction after issue. The NPRED field initially holds the number of predecessors of a node that have not been issued. When it goes to zero, the instruction is added to the issue candidate list and the field is used to point to the next instruction on the list. This is done in AIL. When an instruction is selected for issue it is removed from the issue candidate list and the NPRED field now holds the register number (0-27B) that the *RI* (result) is in.

[[] PAGE BREAK-- end-17-1

Note that since the operand R-numbers point back to the instructions that define the result no search is necessary to look up the operand R-numbers.

SVL holds the saved link words for short instruction sequences so BDT doesn't have to be called again in case of a failure in scheduling mode (because the delta T is too big).

AXR is a bit strip of the available X & 8 registers in the order X0,X7,X6,...X1, B7,B6,...B0 which search order when looking for an available register.

LXR and ALR are similar in structure.

RBV is a table of the register bit values in S0 register order.

FXRA is a table to translate the normalization count of a X-register bit value into the register number, and bit value.

The technique of selecting an available register uses the normalize instruction to find the leading bit set in the available registers in the appropriate class (AXR^CLASS) to translate the shift count to a register number via FXRA.

The normalize instruction is also used to determine the type of result register that is needed for an instruction.

In this case the bits are in the descriptor JFT to RJRS bits ordered so that context dependent situations such as preceding a store or register store take precedence over the basic instruction properties. In both of the above cases the data structures were designed for speed and code simplicity.

RVT is the register value table. RVT(i) points to the instruction that is in Register I. RVT and the REG field of an issued instruction are cross linked and the condition

$REG(TXT(RVTCI)*3) = I$ must hold for I, such that $RVT(I) \neq 0$

PROCEDURES

MCG controller, determine scheduling mode and maximum look ahead

SIS Schedule Instruction Sequence,
call SNI to select the next instruction to be issued,
then call AIL, OUC, ASC, SII to issue instructions & update the issue list,
available resources, clocks, etc.

PRS Preset Storage clear cells, allocate for PIT, OTI, initialize issue list (ICL)
by issuing the BOS

SNI Select Next Instruction from ICL to be issued by finding the first one with the
earliest start time whose late start time is $\leq T+DT$
Calls DRR to determine the result register.

DRR Determine Result Register of an instruction.
Determine result register class from bits set in descriptor & Jump to the appropriate
processor.
Pseudo instructions are processed in the subroutines that DRR calls.
On exit from DRR it has found a result register for the instruction and calculated its
execution times (T issue and T execute), or has not found a register (instruction cannot
be issued now). CRU, FXR, PJI & PPI are subroutines of DRR, & their register usage is
consistent with it,

[[] PAGE BREAK-- end-17-2

CRU Check Remaining Uses of a specified register to see if the instruction under consideration
can be issued now. Called for instructions which PRS.

FXR is both a macro & a subroutine.
The macro is invoked and it calls the subroutine if the register selected is not available
at the current time T, or no result register is available.
In the first case it tries to find the register with the best TRA.
In the second it checks the operand registers of the instruction to see if they can be used
as a result register for this instruction.

PJI/PPI Process the Jump & pseudo instructions.

AIL adjusts ICL to remove the issued instruction from it and to add any new instructions that have become logically issuable to it. It is the heart of the topological sort.

DUC Decrement Uses Counts of the operand registers of the issued instructions and update the register availability List AXR, & RVT.

ASC Advances Simulation Clocks for general instructions & memory references, and changes long memory refs to short ones.

AVC Advances Clock after issue of an unconditional Jump or boundary marker.

SII Save Issued Instruction in PIT in SI. format
by transforming the R-numbers to register numbers
& condensing the 4 word instruction into 1 word OR 2 if a type III with an H2 field.
CLR's are changed into a mask or a BXX-X & SXT's are changed into Boolean or shift XMT.

17.1 JAM MODE - Long/complicated (Dbl Precision ...) clog the registers

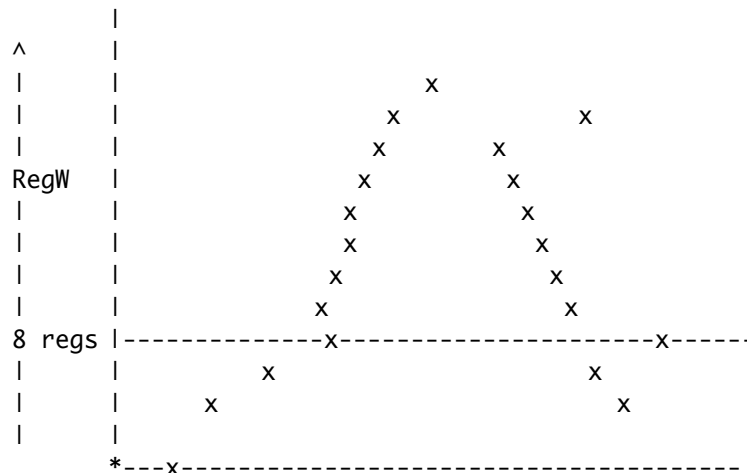
When the basic scheduler fails it calls JAM to code the sequence.
JAM assumes the IL sequence is intact and can be modified.
A description of the strategy appears at MCG.2350.

Since RIO has placed the instruction sequence in a narrow order and computed the number of registers needed at any point, the only question left to answer is what to do when we run out of registers.

Here we have a page replacement problem and we have all the information we need, mainly the list of uses of a instruction (in the successor index table).
Given all this JAM proceeds to issue the instructions in RIO order and create temporaries, reloads as necessary.

[[] PAGE BREAK-- end-17-3

In JAM mode we go through the instruction issue cycle as in MCG (SNI, AIL...SII) and in addition we keep track of the next use of the active registers (SNU) after every time a register is used and not cleared we check its next use. If it is far away, then save the register at that point.



definition	1st use	next use	inst no.
(2 uses)	create temp/reload at this point		

Data structures for JAM mode are somewhat more complex
 RVT holds the TXT index of the next use of an instruction
 and the late start times of an instruction are its TXT index over 2.

JAM mode subroutines (QUAL/JAM/,listing order)

SNI identical to scheduler mode SNI except that the look ahead is very limited.

ISI issues a specified instruction if a result register can be found, calls AIL,OUC,ASC,SII,SNU

CNU Check Next Use of operand registers of issued instruction to see if they should be released now because their next use is "far away". By releasing the registers now instead of later in a crisis situation we may avoid first order conflicts. On the other hand, due to the local nature of CNU we may release a register that is expensive to restore, while if we had waited we might find something cheaper.

SSR save result in specified register (called by CNU)•

SUR stores away results that have been marked to be stored out when the store registers were tied up.

[[] PAGE BREAK-- end-17-4

UJR UnJam Registers when a crisis situation occurs (SNI cannot find an issuable instruction.) UJR frees up a register so the specified instruction can be issued. It calls ORT, CSR, CRA, FRR, ISI, RII, SRM.

RII Reset Issued Instruction (LD or type II) with -no operands- for reissue.

CSI Check for possible issue of a store instruction so as to free up a store register.

ORT determine result register class (type) needed to issue an instruction.

FRR Find a Result Register in a specified class that may be freed up so the instruction can be issued.

CRC Compute Restore Costs of establishing a result in a register and exit with the number of the register whose restore cost is the least and the furthest distance to its next use.

CPA Change Register Assignment. Called by UJR when an X-register is available but it is not in the desired class.

CRA issues an XMT to free a register in the requested class.

SXB Setup XMT Back to original register when the result in the register freed by CRA is PS or PRS.

OXI Output a XMT Instruction to move a result from one register to another.

RIL Reset Issue List to remove an issued instruction from the registers and place it back on the issue list. RIL is the inverse of AIL.

SMB set MUC, multi-use computation bit for instructions that are computations and whose uses are spread out. These instructions will probably be stored into OT.'s and should be placed in store registers if possible so as to avoid an XMT when they are stored.

SNU set next use of operands of an issued instruction updating the NU field in RVT. Scan successor lists of operands to find next use of operands & update RVT. Add issued store instructions to the issued store chain & update the last result in B-reg table.

SRM save a result in memory and set for reload of it. Calls GLI, ISI, and OTS.

OTS Output Temporary Store Call GOT to get an OT. Cell and OEI to issue the ST instruction.

GOT Get OT. cell. Scans OTI for a location that is not in use.

[[] PAGE BREAK-- end-17-5

GLI Generate LD of saved result, & possibly an XMT to move the result to a store register.

OEI Output Extra Instruction, calls ASC and SII.

AIS Allocate Instruction Space for extra instructions when all the extra space preallocated in PRS is used up.

AIP Adjust PIT pointers, reset B7.

[[] PAGE BREAK-- end-17- 6

18.0 SQZ - REMOVE COMMON SUB EXPRESSIONS

SQZ is responsible for
 compile time evaluation of constant subexpressions,
 reduction of algebraic identities such as $0*X=0$, $X+0=X$, $0/X=0$, etc,
 elimination of redundant (duplicate) and useless instructions,
 and removal of redundant temporaries generated by the global optimizer.

On entry to SQZ the sequence to be processed is in TXT, the link words are zero, the uses & precedence fields in the descriptors are zero.

On exit the link words are cleared,
 the USES, STRS and RJRS fields are set in the descriptors
 THE R-numbers are in "canonical order":
 result R-number (RI) = index of the instruction in TXT.

SQZ has three entry points:

SBB - to renumber the »IL* instructions prior to squeezing (called from PROSEQ).

SQZB - which is called by GP0 to squeeze a sequence

SIE - which is called to GPO to squeeze a variable increment expression.

During the entire algorithm RND is used as a scratch table to hold the properties of the instructions and their addresses.

After initialization which is unique for each entry point, they all join for common processing at SBB5A.

The squeeze algorithm consists of four passes over the instruction sequence plus a call to RNI which is two passes

The SBB entry point makes an additional pass to initially renumber the instructions as Proseq has expanded the Macro input from Pass one of the host compiler.

The four passes are:

1. Forward pass to setup RND, back chain the store instructions & chain the RS instructions. A pass over the RS instructions to set the PRS or RJRS bits.
2. A forward pass to eliminate redundant instructions, evaluate constant subexpressions, and collect uses counts for the non-redundant instructions.
3. Backward pass to mark the useless instructions as dead and decrement the uses counts of their operands.
4. A forward or backward pass to squeeze out the dead (redundant or useless) instructions by physically compressing the sequence.

Finally RNI is called to bring the R-numbers to canonical order.

[[] PAGE BREAK-- end-18-1

During pass 1 RND is setup to hold the definition address of an instruction which defines a result (RI), and the RI or R-number it defines.

RND(RI) points to the instruction which defines the R-number RI.

During pass 2 the uses field and property bits are filled in RND, and the R-number field of a redundant instruction is set to the value of the master.

When an instruction with operands is encountered its operand R-numbers are adjusted to account for any elimination of previous instructions that took place by fetching the current values from RND and substituting them in the instruction.

Instructions are marked as dead by setting the descriptor word of the instruction to +0, or -0 in the case of a ST or RS instruction.

The information setup in pass 1 allows us to look at the operands of an instruction, properly squeeze instructions which PS or PRS, and eliminate ST/ST combinations.

Because the CYBER machines are not single register, & because of the subdivision of X-registers into load and store registers the equivalence between two identical instructions is a XMT instruction.

If both have identical precedence bit settings then the XMT is unnecessary.

As an example, consider the statements:

```
Y = A*B
Z = SIN(A»B)
OR
Z = X**X
```

CCG constitutes the majority of the code in SQZ, and we discuss it now.
The order of precedence is:

- a. Evaluate an instruction (constant operands)
- b. Simplify or reduce to a cheaper instruction
- c. Eliminate instruction as redundant (search for a previous occurrence)

Instructions are processed by individual processors as they are encountered

[[] PAGE BREAK-- end-18-2

Non-redundant memory references and Jumps are kept on a backward linked chain, MRC.
When a new memory ref is encountered, its operands are adjusted, and the chain, if non-empty, is searched backward until a ref with an identical IH (R2) word is found or an interfering ref is found, or end of chain is encountered.
In the case that no match is found the ref is added to the chain and the uses counts adjusted.
If a match is found, there are four possible combinations -

LD/LD - the second load is redundant.
ST/ST - the first store is to be eliminated.
LD/ST - eliminate store if LD will be eliminated in EDD.
ST/LD - change LD to an XMT and try to eliminate it.
The case of a ST/ILD is a special one introduced to speed up OPT=2 compilation.
Not only is the ILD changed to an XMT, but the ST is eliminated.

Processing of the Jumps & boundary markers consists of clearing MRC, etc.

Type I instructions are not kept on a chain. Since a redundant occurrence of one must occur after the definition of both their operands, the search range can be narrowed sufficiently to make a simple linear search efficient.

The constant shift instructions are chained for searching purposes,
as are the miscellaneous instructions (STT,S,CLR,FMA).

Because SQZ was the first program written using the algorithmic commenting language it is somewhat difficult to read. The main problem is that the RJ,RK words of a type I instruction are called RJ,RK instead of RJW (=RND(RJ(R1))) and RKW, etc.

The code in SQZ is fairly tight and care should be taken
not to destroy any of the pass 2 dedicated registers (X0, X4, X5, B1, B2, B6, B7).

[[] PAGE BREAK-- end-18-3

19.0 BDT - Build Dependency Tree

BDT contains routines to form the dependency graph (a network) &
to reorder an instruction sequence.
The current version represents the fourth version (rewrite) of BDT.

The deck BDT contains four entry points -

BDT - which forms the PERT network of an instruction sequence.

RIO - reset instruction order, which reorders an instruction sequence so that the instructions are in a narrow order to minimize register requirements.

CRW - Compute Register Width of an instruction order.

RNI - renumber instructions, which brings the R-numbers to "canonical order".

BDT forms the PERT-Time network of an instruction sequence by treating the IL instructions as activities, the execution time of the instruction is the activity time, and the operand & logical links between the instructions are the activity constraints or edges of the graph.

On exit from BDT the successor index table has been formed in TREE, and the link words of the instructions point to the successor list in TREE, etc.

During the first phases of BDT the temporary tree is formed in TREE, starting at the end of TREE and working towards the beginning. The equivalence (data interference) links are formed first since this phase may be the most space consuming. Note that a sequence with K memory references may have up to $(K**2)/2$ equiv links.

FIL forms the *IH* info table starting at 0.TREE by a forward scan of the sequence. Then the IH info table is scanned for interfering parts of memory refs. (FIL3), and when one is found an edge is added to the tree.

Next the edges that correspond to operand links are formed in a forward scan of the instructions (FOL). This phase also forms a list of the boundary marker & -sink- (stores & RS's) instructions for the next phase.

FTL forms the terminal links by a backwards scan of the BM list that FOL built. It also forms redefinition links for the predecessors of RS's when two or more RS's occur in a basic block. At the end of FTL the temporary tree is sorted to group the edges by successor. This is done to simplify the PERT time calculations. FJL forms logical links from a conditional Jump to an instruction that appears after it, but that has no operand links to previous instruction in the same basic block. For example, consider

IF(X.LT.Y) Z = X*Y

[[] PAGE BREAK-- end-19-1

where the multiply must be linked to the Jump to prevent it from being issued before it (prevent spurious mode errors). CIP calculates the instruction late start times, that are computed by the formula -

late start time (i) « totime - neg late start time (i)
where totime = neg late start time of the BOS.

In JAM mode the late start times are setup as the TXT index of the instruction over 2, and duplicate edges are eliminated from the tree for MCG/RIL.

SIT reformats the tree as a successor index or uses index table by shifting the position of the SUCC and PREO fields and resorting the tree. Finally the index to the SUCC lists in TREE are setup in the link words of the instructions and BDT exits.

GAS is entered by any one of the phases in BDT when it thinks that there is not enough storage to build the tree. Most of the time this happens when there is excessive data interference.

After allocating extra space it resets the registers and starts all over again

19.1 RIO - RESET INSTRUCTION ORDER

RIO reorders the instructions to bring them to an order that is "minimum width" with respect to its X-register requirements.

The algorithm used is straight from Aho-Ullman Volume II which is no longer available and achieves a minimum only if the sequence contains no multiple uses.

Chapter 9 of "Compilers..." by Aho-Ullman covers Register assignment.

After a sequence has been processed by RIO it is said to have been -JAM ordered-.

The information collected by RIO is used by

1. MCG to decide what order to issue instructions in JAM, and the reg widths are used to determine when to unload a register.
2. AIS to determine when it may be profitable to move a result to a 3-register.
3. GRA to determine if X-register assignment is possible and how many can be assigned.

The general problem of Jam ordering a sequence is was and probably still a Thesis topic. Wuttf suggests an extension of the Aho-Ullman algorithm in his article on Bliss, which is worth looking at.

In order to compute the new order for the instructions, RIO calls BDT to form the TREE (without templock RS redefinition links), computes the node labels or weights (number of registers needed to code the instruction and its predecessors), reformats the tree and sets up the link words.

[[] PAGE BREAK-- end-19-2

SIO uses a depth first search of the dependency tree, whose* edges have been sorted by the node level derived in RIO, to develop the new instruction order.

It operates in one of two modes, general reordering, or with a subsequence demarcated by sink instructions. Because of difficulties in the initial design, the activation of general reordering mode had to be postponed, and the other mode was created.

The initial problems that caused the postponing of it were the lack of variable redefinition links in the tree, and the fact that it would separate stores from their definitions (RI). The lack of redefinition links for memory redefinition links for memory references would allow the uses of scalar variables to extend past a redefinition of it, and this would cause incorrect code, or deadlock situation if GRA made any B-register assignments. The other problem would cause spurious XMT/SA instructions to be generated by GRA.

In general reordering mode,

BDT adds variable redefinition links to the tree.

RIO forms chains that link from store predecessors to its stores.

SIO when encountering a ST predecessor, adds its ST's to the stack if necessary to ensure that they will follow the predecessor. It also makes a special check for deadlock situations (see example in section on deadlock situations), and in this case will separate the store from the definition of the RI in the case that the sequence is going to be processed by GRA.

After deriving the new order, SIO copies the instructions to TREE and moves them back to TXT in their new order. It then calls RNI to adjust the operand R-numbers and CRW to compute the register width of the order.

CRW - computes the X-register width of an instruction order at each instruction in a sequence and the max width of the sequence. The -REGW- of an instruction sequence at an instruction is defined as the number of results that are in X-registers whose uses have not been exhausted. The REGW's are computed by forward scan of the sequence, followed by a forward pass to clear the link words of the instructions.

19.2 RNI - RENUMBER INSTRUCTION R-NUMBERS

RNI transforms the operand R-numbers in a sequence to bring them to "canonical form", that is if the instruction has an RI which defines a result then it is given a new value equal to its index in TXT.

R-numbers are assumed to be in canonical form in all parts of pass 2 except PRE, SQZ and parts of PROSEQ.

RNI consists of two passes over the instructions, the first one forms the equivalence table such that $TBL(Old\ Ri) = New\ Ri - Old\ Ri$ and the second pass installs the new R-numbers. Because of RJRS's RNI needs a second pass.

[[] PAGE BREAK-- end-19-2

20.0 CCG UTILITIES & ROUTINES TO FORMAT DEBUGGING OUTPUT

20.1 CFA - CONTROL FLOW ANALYSIS

CFA processes the control flow information collected by the Bridge to form the graph structure tables, which consists of the interval lists and the back dominator information.

Interval Analysis as a method of determining the extent of strongly connected regions was invented in the late 60's by John Cocke & Fran Allen at IBM. As a grad student at NYU/CIMS (1965-68 & 70-72) I became familiar with the method. By the 1980's it had become well known and is discussed in "Compilers..." by Aho-Sethi-Uttman among other texts.

It is the theoretical basis of the algorithms used in CFA.

Over the years different groups have developed various algorithms for processing a program flow graph to find the loops, back dominators, etc.

The interval method was picked for use in the compiler since the author had some previous experience with it and it appeared to work reasonably well.

In the process of implementing it for the compiler various problems, never mentioned by Allen et al, were discovered and corrected, as discussed below.

In the compiler a modified interval method is used to find the loops in the program, and to direct the global optimization, which consists of an inner to outer pass to move out invariant code, strength reduce integer expressions and register assignment of scalar variables. The differences between the Interval Method as proposed by Allen & Cocke and as implemented in the compiler are

1. No outer to inner pass

2. No global expression dictionary

3. Only the strongly connected parts of the intervals are processed during loop optimization and then reduced to a point, instead of processing and reducing the whole interval to a point. This is done because Intervals may have a long -tails- which may be unrelated to the loops within the interval.

4. The node splitting multi-entry loop process is bypassed by terminating the derived graph development process when the current graph has no loops.

Problems encountered with the interval method were

1. Loops without exits. This type of loop occurs in some odd cases, and was a nuisance case to handle.

2. Order of the nodes within the intervals, while far from random, was not always program order, in the case that the interval contained an extended block and an "error exit", the breadth first nature of the interval formation process tended to get the nodes out of order. This necessitated sorting the nodes within the interval after it was formed.

[[] PAGE BREAK-- end-20-1

3, In order to form the intervals in program order the interval header list was searched for the smallest node number in it. This is done in order to minimize the number of page faults in BLK when processing small innermost loops,

4. Extensive fiddling around with the node numbers had to be done in the derived graph processor so as to keep the problems mentioned in 2 and 3 from arising.

SUBROUTINES

CGT - Check Graph Table size prior to exiting CFA & flush them to disk if they are taking up too much space in memory,

DGS - Derive Graph structure. This is the main entry point to CFA.
AFT is called to adjust the flow table to remove references to symbols,
FGS is called to form the graph structure tables,
FIS forms the interval lists and computes the back dominators.
MRB checks the graph for dead code and issues messages if there is any.
The processing continues to form the derived graph of the interval,
the graph structure tables and interval lists of the derived graph
as long as the current graph has loops.
On exit the graph tables are in TXT, and may be on disk.

AFT - Adjust Flow Table (CFT) to replace the *IH* ordinals,
from forward references, with node numbers.
Scan CFT for edges where the T0 field is an IH & replace it with the node
number of the block.
The node numbers of blocks with labels are kept in bits 0-11 of word B of
symbol table entries and bits 0-17 of the GL table,

MRB - Mark Reachable Blocks,
Scan interval lists and set RB for blocks that are reachable.
If all blocks are reachable then exit,

else reform CFT to eliminate the dead nodes from the graph and call FGS
& FIS to reform the graph structure tables and interval lists.
Finally issue messages to the Dayfile (6600 display console)
& list file about the dead code.

FGS - Form Graph Structure Tables.

Reformat CFT form successor & predecessor lists, the edge index (EI.) & edge tables (ET.)

FIS - Form interval structure lists of a program flow graph.

This routine is subdivided into two major parts,

FIS where the intervals are formed & CBO which computes the back dominators.

The basic algorithm is that described in Aho-Ullman with embellishments.

During the interval formation process participating blocks are given -node numbers-

from 1 to the number of blocks in the graph. For the first graph node num = block num/2.

Created nodes (loop prologue) are given the next available block number, & node numbers
in FDG when the derived graph is formed.

FDG - Form Derived Graph of the interval structure of a graph.

The SCR of the intervals are reduced to points & the resulting graph is called
the derived graph.

The nodes are renumbered & the edge table formed for FGS.

[[] PAGE BREAK-- end-20-2

20.2 UDT - USE- DEFINITION TABLE PROCESSING

UDT contains miscellaneous routines for the pre-processing phase of OPT=2.

Except for PBB, the function of the routines is to form the Use/Def table
which is used during the global optimization phase.

During processing of a region the Use/Def table serves as a repository for the USE/OEF
information associated with each memory reference. The entries in UDT represent a single
memory location (CA,IH) or a class of memory locations that may be referenced by an
indexed memory reference (IH only).

Examples of element references are A(4), X, Y, and A(I), A(J),

call ZZ(A) are examples of class references (which all represent the same class).

During the preprocessing phase all unique reference classes appearing in the program are
entered in UDI, and the IL instructions, AP and I/O lists elements are chained to it.

In order to speed up formation of UDI, a simple hash procedure with an auxiliary base
table (UDB) is used. At the end of the preprocessing phase AUT reformats UDI and chains
class members (A(2)) to the class representative entry. Each class is chained together
from the class representative so that a definition can be propagated to the class.

SUBROUTINES

AUT - Adjust Use/Def Table to bring to UD-format. Form a sort table from UDI and sort by IH.

Reformat UDI, chain class members to class rep, etc. Form word index,

bit number values for each entry and common variable -spoil- bit vector.

Form special bit vectors for use in live-dead and Iysis.

CMR - chain memory references in IL sequence to UDI.

Scan sequence for memory references, search UDI for entry or make a new one, chain ref to UDI by placing ordinal in IN field of the RI word.

CPL - Chain Parameter List entries (AP and I/O lists) to UDI.

PBB - Process Basic Block for OPT=2.

Called by PROSEQ after the block is squeezed to chain it to UDI, append the I/O lists to the end of it, and to write to mass storage and create a BIT index for it.

[[] PAGE BREAK-- end-20-3

21.0 CCG INTERFACE SPECIFICATION APPENDIX

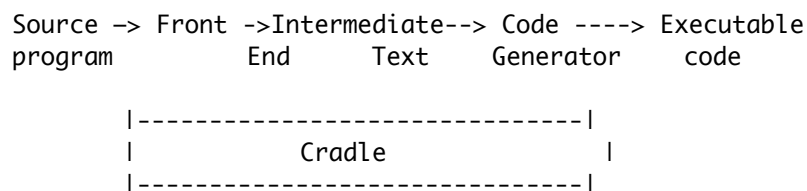
SIJ NOTE - As CDC is dead as a company and it's computers are emulated on today's Laptops at faster speeds than the than the monster mainframes ran at, this section did NOT get the extensive error checking that the previous chapters did.

1.0 INTRODUCTION

This chapter provides an overall picture of the structure & functions of the code generator and provides a description of the structure of this document together with notation conventions which are used in it.

1.1 COMPILER COMPONENTS

The basic parts of any compiler may be described as a front end, a code generator, and a cradle, as depicted in the figure below.



The function of the cradle is to provide an environment for the compiler including communications cells, service routines, overlay loading etc.

The front end parses the input stream, performs syntax and semantics checking, supplies defaults, completes data descriptions, resolves references, makes all operations explicit, simplifies where appropriate, may perform some language dependent storage mapping, & outputs an intermediate text which is expandable into machine instructions & pseudo operations.

The code generator performs optimizations (strength reduction, dead code elimination, common subexpression analysis, loop analysis, etc.), allocates machine resources including registers and functional units, issues executable code and performs the final assembly.

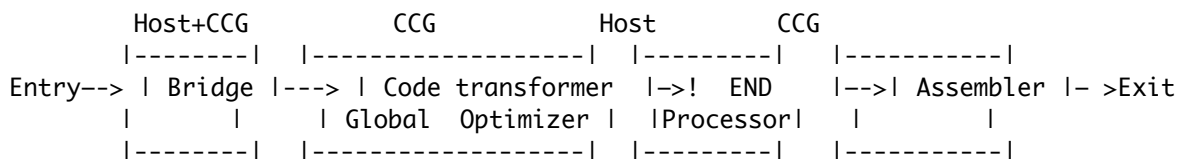
[[PAGE BREAK-- end-21-1

CODE GENERATOR COMPONENTS

The code generator contains the package referred to as the Common Code Generator (CCG), but code generation is accomplished as a cooperative effort between CCG and routines supplied by the host compiler.

CCG may be considered to consist of two parts: the - CodeTransformer and the Assembler. The Code Transformer, in turn, consists of two parts: the Sequence Optimizer which is always executed, and the Global Optimizer which is executed when the appropriate (OPT<2) optimization level is requested.

The host supplies two sets of routines to interface with CCG: the Bridge and the End Processor. The flow of control through the code generation process is depicted below.



CONTROL FLOW

The Bridge gains control and drives the Code Transformer by repetitively calling the Sequence Optimizer to hand off code sequences. When the entire source program has been processed, the Bridge gives up control and the Global Optimizer takes over (OPT=2 only). After the Transformer is through, the End Processor is given control. Following the End Processor, the Assembler finishes the code generation process.

The flow of data through the Code Generator is as shown in the following diagram.

[[PAGE BREAK-- end-21-2

https://www.jasik.com/CCG_IMS_ISA_3.jpg

[[PAGE BREAK-- end-21-3

The SLIST file is used to accumulate the entire source program represented in the language required by the Assembler.

The Front End and/or Bridge write initial declaratives, the Code Transformer writes the machine instructions and the End Processor writes the final declaratives.

Symbol Table BR and the Intermediate Text is strictly a function of the Front End.

From these inputs, the Bridge must provide Symbol Table CT, whose format and content is as required by the Code Transformer.

The Bridge must also segment the program into sequences (blocks), translate them into the Internal Language of the Code Transformer & present them one at a time to the Code Transformer.

From these inputs, the Code Transformer performs optimizing transformations on the code sequences & provides additional information in the symbol table to bring it to the state of Symbol Table EP. The End Processor must provide any information which is required to bring the symbol table to the state of Symbol Table AS, whose content and format is as required by the Assembler.

The decks comprising the Code Generator may reside in one or more overlays.

The restrictions are that if it resides in two overlays, then the assembler (CGIA and MACROS) may reside in a secondary overlay and the deck CGTM must reside in a main or primary overlay that is common to both CCG overlays. If the host supplies his own assembler then CGTM resides in the overlay with the rest of CCG.

The host provides the control routine. The control sequence must be as follows:

1. Call the Bridge. The Bridge, in turn, calls the Code Transformer●
2. If OPT-2, call the Global Optimizer.
3. Call the End Processor.
4. Call the Internal Assembler (CGIA).
5. If the cell N\$FERR is $\neq 0$, it contains the number of fatal errors encountered.
An appropriate message should be issued.

Three entry points should be provided within the control routine. If the Code Transformer, the End Processor or the Internal Assembler irrevocably runs out of memory, it will exit by an EQ jump to the appropriate entry point.

These entry points are:

- HE\$CTX for Code Transformer
- HE\$EPX for End Processor
- HE\$IAX for Internal Assembler

[[] PAGE BREAK-- end-21-4

1.4 STRUCTURE OF THIS DOCUMENT

From the preceding discussions, it can be seen that there are four operationally distinct sets of routines in the code generator:

- The Code Transformer which is discussed in Chapter 3.
- The Bridge which is discussed in Chapter 4
- The End Processor which is discussed in Chapter 5

The Assembler which is discussed in Chapter 6.

It can also be seen that there are three «alor data structures:

- ° The symbol table and associated tables.
These tables persist throughout the code generation process. They are presented in Chapter 2.
- ° Code sequences.
It is the primary function of the Bridge to produce code sequences,
Their description is in Chapter 4
- ° The SLIST file. Although this file pervades the entire code generation process, a comprehensive discussion of it requires a discussion of the Assembler.
It is delayed until Chapter 6.

The remainder of the document is devoted to peripheral subjects.
Chapter 7 describes the requirements that CCG imposes on the Cradle.
Chapter 8 discusses the operational environment,
and the compiler build environment is described in Chapter 9.

A concerted effort has been expended to structure the document so that it may be understood by reading it straight through. However, the reader should be aware that a full understanding of some subjects cannot be realized until chapters later in the document have been read. Forward and backward references are provided for such cases to aid the reader in the inevitable cyclical understanding process.

1.5 NOTATION CONVENTIONS

Certain coding conventions which are used in CCG are also used in this document as notation conventions as described in the following sections.

1.5.1 Table Description Notation

The following macros are used in this document to specify table structures.
They are the same macros which CCG expects the host compiler to use to set up symbol definitions.

[[] PAGE BREAK-- end-21-5

DESCRIBE, DEFINE, REDEF, DEQU MACROS

These macros facilitate field description, where fields are sub-word entities.
The DESCRIBE macro is used to provide a prefix for names supplied in subsequent DEFINE, REDEF, & DEQU references. For each name specified, the following symbols will be defined:

PFX->NAME-P – the bit position of the right most bit which comprises the named field
(within a word bits are numbered according to the power of two which they represent)

PFX->NAMEi-L – the length in bits of the field. Fields are not permitted to span word boundaries
(a fatal-to-assembly error will result) or to go beyond the total number of bits which
the entire set of fields is supposed to occupy.

The definitions of these macros are contained in the comdeck COMADEF,
which is called by CMPLTXT & CCGTEXT.

DESCRIBE reference

OESCRIBE PFX,BITSLONG,TOPBIT where

PFX is the prefix mentioned above.
BITSLONG is the total length in bits of the structure.
If not present a value of 60 is assumed.
TOPBIT the beginning (i.e., leftmost) bit of the structure.
If absent *BITSLONG-1 is the default value (59).

DEFINE reference

NAME OFINE LENGTH where

NAME is the name of the field mentioned above.
If not present, then the reference acts as filler

LENGTH is the bit length of field (or filler).
A value 1 is assumed if this parameter is omitted.

Each OEFINE is considered to reference a field beginning to the immediate right of the preceding field.

[[] PAGE BREAK-- end-21-6

REDEF - declare overlapping substructures

REDEF NAME where

NAME is the name of a previously declared field.
The next field to be defined begins at the leftmost bit of the field NAME.
If NAME is omitted, redefinition begins at the top of the entire structure.

DEQU - define an &equival9Pcga f i&ig. cof sctQGa

AA DEQU BB,LEN where

BB is the name of a previously defined field
AA is the name of the field being defined

LEN is the length of the AA field, if omitted the length of BB is used.
The bottom bit of AA is the bottom bit of BB.

(say for TYPE /g 1001

Consider the following examples:

TABLE X	VFD 12/TYPE,30/VALUE,18/SUBV
OESCRIBE X.,60	TABLE X, the prefix to be used is the two characters x.
TYPE OEFINE 12	
MSG OEFINE 48	
REDEF MSG	
VALUE OEFINE 30	
SUBV OEFINE 18	

The symbols defined wo«

X.TYEP	EQU	48
X.TYPEL	EQU	12
X.MS5P	EQU	0
X.MSG1	EQU	48
X.VALUEP	EQU	18
X.VALUEL	EQU	30
X.SU3VP	EQU	0
X.SUBVL	EQU	18

[[] PAGE BREAK-- end-21-7

In this document, references to a particular format are made by referencing the prefix from the OESCRIBE macro.
For example, if the format of table X above were to be specified, it would be referred to as "X." format.

1.5.2 NAMING CONVENTIONS

Since CCG is intended for use by any number of host compilers and thus has the potential for name conflicts, a naming convention has been established which applies to the assembly time and load time environment.

The special character E (#) is reserved for internal symbol use by CCG.

A set of standard comdeck entry points is used, the names of which terminate with =.

Every interface symbol (with the exceptions noted below)
has a name conforming to one of the following classes.

HC\$XXX	Host compiler assembly options (symbols and micros)
HO\$XXX	Host compiler option cells
HE\$XXX	Host compiler entry points called by CCG
CG\$XXX	CCG procedure entry points
CC\$XXX	CCG communication cells
O\$XXX	Contains table origin
L\$XXX	Contains table length
Z\$XXX	Table ordinal or other small assembly time constant
F\$XXX	Absolute table FWA
N\$XXX	Address of a cell containing an integer constant
M\$XXX	SMACRO opcodes (not referenced by CCG)
S\$XXX	Contains a symbol table ordinal of a special symbol

The following names, which do not conform to the above conventions,
are reserved for use in the CCG interface.

Symbols

R1.XXX R1 word describe
R2.XXX R2 word describe
IH.XXX IH word describe
D.XXX Descriptor word describe
WA.XXX Word A describe
WB.XXX Word B describe
WC.XXX Word C describe
CP.XXX C0MPC0M symbols
OC.XXX IL opcodes
JC.XXX Jump codes

[[] PAGE BREAK-- end-21-8

SI.XXX

AP.XXX
 CF.XXX
 FI.XXX
 F.xxx
 SO.XXX
 IP.XXX
 .CSET
 .DS
 .IWT
 CT.ECS
 CT.CPU
 .CPU
 L.STACK
 .DAL CP=RM OTrRM ADH\$ AT\$ PRNTXXX HMPXXX CCGXX

Micros (CHAR STRINGS)

OS.NAME OS.VER MDL\$ SCM\$ LCM\$ LPN\$ VER\$ MODLVL\$

COMPASS Ident & Update Deck Names

CGIA MACROS CGTM MIO FBV GPA GPO PROSEQ SQZ BDT MCG CFA UDT
 PRINTMIO PRNTMCG PRNTGRA PPNTRLI PPNTUDI PRNTABV DMPIIT OMPRLST DMPSIT DMPTREE DMPUDI

[[] PAGE BREAK-- end-21-9

?????

SLIST describe APLIST describe Control flow describe Function information describe FETS
 Register designator describe Installation parameter

[[] PAGE BREAK-- end-21-10

2.0 THE SYMBOL TABLE & ASSOCIATES

The symbol table and the tables that are associated with it persist throughout the code generation process. This chapter presents the specification of:

- The main symbol table - SYM
- Abbreviated symbol tables • GLT, APT, etc.
- Block tables - CBT, LBT
- Constant tables - CVT, CUT

2.1 SYMBOL TABLES

CCG requires a main symbol table, SYM, an abbreviated table, GLT and also permits other abbreviated symbol tables for certain classes of symbols.

2.1.1 The Main Symbol Table - SYH

Each symbol table entry is 3 words long.

The cells O\$SYM & L\$SYM hold the FWA & length of the symbol table.

The symbol table grows from RA towards FL.

Entry 0 is a dummy entry for the use of CCG! it contains three words of binary zero.

The symbol table may not grow during CCG processing (after CG\$INIT is called).

The general use of the words in CCG is:

Word 1 (Word A) - BCD name word

Word 2 (Word B) - Symbol attributes word

Word 3 (Word C) - Address definition information word

Symbol table formats are language dependent and CCG allows, within limits, the host to place fields at arbitrary bit positions within Word B.

The following sections describe the format & usage of the fields within the three words.

Each field definition has a parenthetical list following the OEFINE macro.

The meaning of the items in this list are:

CT The field value is supplied by the Bridge & used by the Code Transformer

EP The field value is supplied by the Code Transformer
and, hence, available to the End Processor.

AS The field value is supplied by the Bridge or the End Processor & is used by the Assembler.

[[] PAGE BREAK-- end-21-11

2.1.1.1 word 1 (Word A) Format and Usage

OESCRIBE WA.

NAME OEFINE 6*HC\$MCIS (AS) (48 bits of 0L display code followed by a binary 0)
6-bit display code name of the symbol in *QL* format. It must be a legal COMPASS name.
The Assembler will append a \$ to any name that conflicts with a register name.

8N DEFINE 12 (CT,EP,AS)
used for internal purposes by the Code Transformer & the Assembler.
The End Processor may use it, but the Bridge may not.
Its initial setting is not pertinent.

2.1.1.2 WORD 2 (Word B) Format and Usage

DESCRIBE WB. The following Word one bit fields may occur in any bit position (??)

LAB OEFINE 1 (CT,AS)
= 1 if the entry is for an internal transfer label which does not have any of the properties represented by the fields normally found in the rest of this word.
When it is set, neither the Code Transformer nor the Assembler depends on the setting of the rest of the word with the exception that the Code Transformer uses WB.LC for internal purposes and expects it to be set to 0 initially.

LC DEFINE 1 (CT)
This bit is used by the Code Transformer if LAB = 1. It must be initialized to zero.

MAT DEFINE. 1 (EP)
The Code Transformer sets this bit if it sees a memory reference to the variable after it has performed all of its optimizing transformations. It is intended for use by the End Processor to eliminate storage associated with variables that are not materialized.
The host is responsible for setting this bit for references that the Code Transformer does not see (e.g., actual parameter lists).

LDO DEFINE 1 (CT) (LOAD ONLY)

= 1 if all references to the symbol are loads. It is used in OPT=2 to reduce table space

[[] PAGE BREAK-- end-21-12

LOCF OEFINE 1 (CT) (Address of)

This bit is set if the variable may be referenced by some method which the Code Transformer cannot be aware of. It is used to inhibit certain optimizations when OPT-2 (dead definition elimination & invariant code motion, described in Section 3.1.4) Examples of such a variable are!

a. X = Z
 Y = LOCF(X) (LOCF Obtains the address of it's argument)
 CALL USER(Y)

If the LOCF bit were not set for X, the Code Transformer would eliminate the statement X = Z because there are no apparent uses of the definition of X. It does not see the use of X in the Y=LOCF(X) statement) because it does not appear in one of the load or store instructions of the Internal Language (LD, ST, PLO, PST), described in section (4.3.3).

b. The FTN compiler handles the returns value for a function by storing the value in VALUE, & on exit from the function it loads VALUE into X6. However it handles the exit code as a canned macro which is never passed through the Code Transformer. Hence, the Code Transformer never sees the final use of VALUE, and would probably optimize out the stores to it, if the LOCF bit were not set.

WEXT OEFINE 1 (AS) (Weak External)

This bit is set to one if the entry is for a weak external
When this bit is set the EXT bit must also be set.

The following one bit fields must occur in the listed order.

LCM DEFINE 1 (CT)

= 1 if storage for the variable resides in LCM (not ECS). It is part of the definition of the address expansion type for FORTRAN (see AET field below).
For languages with a different AET, this bit must be set to 0.

FP OEFINE 1 (CT,AS)

= 1 is a formal parameter. It is part of the definition of the address expansion type for FORTRAN (see AET field below). For languages with a different AET, this bit must be set to 0.

COM DEFINE 1 (CT)

= 1 if storage for the variable is in a common block.
It is used to inhibit certain optimizations (dead definition elimination, invariant code motion and address differencing, described in Section 3.1.1*)

[[] PAGE BREAK-- end-21-13

EXT OEFINE 1 (CT,AS)

= 1 if symbol is external. It is used by the Code Transformer to inhibit address differencing.

ENT OEFINE 1 (AS)

=1 if the symbol is an entry point.

The following fields may reside anywhere in Word B.
Some of the field lengths are specified with a question mark.
It is at the discretion of the host compiler to specify these field lengths.

- FPO OEFINE ? (CT)
Contains the formal parameter ordinal. It is used in conjunction with the FP field to make indirect references to formal parameters via the actual parameter list when FORTRAN memory references are expanded. See Section 3.2.2 for further discussion.
- AET OEFINE ? (CT)
address expansion type. It defines the address mechanism that must be generated by the Code Transformer to access the base address of a memory reference. The meanings of the values that the field may take on are encoded in CCG as host compiler dependent code that is conditionally assembled. If there is no meaning to the field (i.e., no address expansion at all), the field itself must not be defined.
See Section 3.2.2 for further discussion.

2.1.1.3 Word 3 (Word C) Format and Usage

- DESCRIBE WC.
- RL OEFINE 2 (CT,EP,AS)
specifies the relocation for the symbol as follows:
 0 - ABS symbol
 1 - Program relocation
 2 - Common relocatable symbol
 3 - External (weak or regular)

It must be defined for all symbols that are not labels at the beginning of the CCG pass.

DEFINE 25 (unused)
Use of this field by the host might be allowed upon request.

[[] PAGE BREAK-- end-21-14

- RB OEFINE 9 (CT,EP,AS)
specifies a block number by containing an ordinal to LBT or CBT (see Section 2.2.1).
It must be defined for all symbols that are not labels at the beginning of the CCG pass.
- RA DEFINE 24 (EP,AS)
contains the ABS, program, or common relocatable address if RL≠3.

For internal transfer labels, the Code Transformer sets up the WC. word as:

RL = 1
RB = local block number at time of definition
RA = block relative address.

The End Processor is responsible for changing the block relative address for all symbols to a program relocatable address.

2.1.2 Abbreviated Symbol Tables

Certain classes of symbols are eligible for representation in an abbreviated symbol

table. An abbreviated symbol table entry is one word long containing the address definition information in the WC. format for Word C as described in Section 2.1.1.3 above. The symbol name is mechanically generated to be XX.NNN where XX is the name of the table (see HC\$FRTF, Section 9.3) and NNN is the ordinal of the symbol within the table. Symbols represented in an abbreviated symbol table may not have any of the properties represented in Word B of the main symbol table (except internal label). One abbreviated symbol table, GLT, must exist. It contains compiler generated labels & the Code Transformer makes entries into it Just as it does for Word C of label entries in the main symbol table. Other typical uses for abbreviated symbol tables are AP list & IO list locations. Within each symbol table, the first entry (ordinal zero) is reserved for the use of CCG. Symbol entries begin at ordinal one.

2.1.3 Symbol Table References

As there are provisions for more than one symbol table, a reference to a symbol table contains a table designator, I, and an ordinal into the table, H. IH is used in this document as a mnemonic to refer to symbol table pointers.

[[] PAGE BREAK-- end-21-15

The format of a symbol table reference is: 3/I,3/XX,l2/H where

I is the table designator.

0 - SYM

1 • GL

2 thru 6 - abbreviated symbol tables as specified by HC\$FRTF (Section 9,3)

7 - reserved

XX is reserved for the Code Transformer and must be set to zero initially.

H is the ordinal into the table.

2.1.4 Predefined Symbols

The Code Transformer generates instructions which use the symbols discussed below. Each of these symbols must have an entry in SYM and its ordinal in SYM must be in a Cradle entry point as follows:

Entry Symbol

S\$CON CON The array of constants whose values are in the constant table, CVT (Section 2.2.2)

S\$OT OT. The array of temporaries generated by the Code Transformer scheduler (Section 3.2.1 for further discussion).

S\$IT IT. The array of temporaries by the Code Transformer optimizer (Section 3.2.1 for further discussion)

2.2 ASSOCIATED Tables

The tables discussed in this section are associated in various ways with the

symbol tables. Each was mentioned briefly in the previous section.

2.2.1 BLOCK Tables - CBT, LBT

CCG provides for the use of COMPASS style common blocks & local blocks. Tables in this section are used to record the name & length information about them.

[[] PAGE BREAK-- end-21-16

COMMON BLOCK TABLE - CBT

Records the names and lengths of the common blocks.

It is used by the Assembler only and is not modified by CCG.

Its format is!

60/0Lname

1/R, 35/avl,24/length

where R=1 if LCM/ECS resident, avl=available for use of host.

The first entry in CBT is used for the blank common block. The name must be 7 blanks. If the length is zero, it is assumed that blank common is not used.

LOCAL BLOCK TABLE - LBT

The Local block table, LBT, is used to record the lengths for local blocks.

It contains a one word entry for each local block in the order specified by HC\$FLB (Section 9.3).

Its content changes during the code generation process as is described below.

On entry to the code transformer each entry must contain:

2k/0,18/parcel count,18/length

L\$LBT contains the number of local blocks plus one.

The last word of the LBT is reserved for the use of CCG.

During Code Transformer processing the current block is reflected in the entry points:

CC\$LBO - LBT ordinal of the current local block

CC\$BLEN - current length of the current block

CC\$PC - current parcel counter of the current block (0 to 3)

The End Processor is responsible for bringing LBT to the format required by the Assembler which is:

42/fl,18/first word address (program relocatable address)

The use of this table is discussed further in Sections 3.2.4 & 5.2.1.

LBT is a static table that resides in the host's cradle.

It's origin and length are the entry points F\$LBT and Z\$LBT.

2.2.2 Constant Tables - CVT, CUT

There are two tables associated With constants: the constant value table, CVT, & the constant use table, CUT.

Each entry in CVT contains a one word (load only) constant.

The Code Transformer extends this table.

The Bridge may add entries to it by calling CG\$SCT (Section 8.4.1),

[[] PAGE BREAK-- end-21-17

CUT is parallel to CVT which is used to eliminate the reservation of storage for constants which are not used. Its format changes during the code generation process:

During Bridge/Code Transformer processing each CUT entry signifies whether or not the corresponding CVT entry has been used (non-zero signifies use).

The host compiler is responsible for setting the CUT entries for constants whose use is not seen by the Code Transformer, It may do so by calling the routine CG\$FCU (Section 8.4.1). After performing its optimizing transformations, the Code Transformer sets (to nonzero) the CUT entries for constants whose uses have not been optimized out.

After all constants and constant references have been processed, the End Processor calls CG\$EP (Section 8.4.2) which outputs the used constants to SLIST and changes CUT to contain new ordinals for the constants.

The Assembler modifies constant references by using the SLIST ordinal in a constant reference as an ordinal into CUT to obtain the new ordinal. A special machine instruction, ACC, & a macro directive, AAE, are provided for accessing the constants.

2.2.3 Variable Dimension Information Table - VDI

The VDI must exist if the host is using the LDV opcode.

It is initialized & added to by the host.

Its format is described by the host in the comdeck SYMDEFS,

When an LDV opcode is encountered in the code transfer output stream,

CCG calls the routine CG\$AVO to set the "MAT" bit (D.MATP) in the CA'th entry of VDI & assign a final CA to the reference which is also saved in VDI.

The output processor changes the LDV to a LD as no further special processing is needed.

During end processing the host is responsible for setting up the final value of the CA of the VDI entries in the lower 18 bits of each referenced entry.

During assembly (CGIA) the instruction LDV AI,0,CA,IH is transformed into a SAI IH+K, where K is the lower 18 bits of VDI(CA).

[[] PAGE BREAK-- end-21-18

2.2.4 Formal Parameter Information Table - FPI

FPI is necessary on if HC\$FPAS is non-zero.

FPI contains one word for each formal parameter in a program unit.

The entries contain the symtab ordinal of the F.P. in the PNT field,

the number of SCM address substitutions in the LEN field, & the number of "level 0" address substitutions in the SUB0 field. All of the above fields are 18 bits long.

The LEN and SUB0 fields are incremented by the code transformer.

The FPO field in Word B off the symbol table is an index into FPI.

The host is responsible for defining the FP. symbols [PNT, SUB0 & LEN] in the comdeck SYHDEFS.

[[] PAGE BREAK-- end-21-19

3.0 THE CODE TRANSFORMER

The primary function of the Code Transformer is to perform optimizing transformations on the code presented to it by equivalence-preserving transformations which reduce the time and/or space cost of a computation.

Auxiliary functions that it performs are storage assignment for CCG generated temporaries, address expansion & substitution, assignment of addresses to labels, code length computation & record keeping on the materialization of variables.

3.1 OPTIMIZATIONS

The Code Transformer operates in two distinctly different modes, depending on the optimization level called for by the Bridge: OPT=2 and OPT<2.

When OPT<2, the Code Transformer performs optimizing transformations on one code sequence at a time, completing each sequence as it is presented to it and writing it to the SLIST file.

When OPT=2, the code sequences are optimized as individual units but, in addition, the entire program is accumulated for global analysis and global optimizations are performed. The code is written to the SLIST file only after the entire program has been analyzed & transformed. The general categories of optimizations which are performed are:

- Straight Line Code Optimizations - performed on all code.
- Innermost, Well-behaved Loop Optimizations - performed on demand. OPT<2 only.
- Global Optimizations - performed only if OPT=2.

These categories are discussed below after a discussion of memory reference resolution which is a central concept to most of the Code Transformer's algorithms.

3.1.1 Resolution of Memory References

Virtually every algorithm used for performing the optimizing transformations depends on being able to resolve memory references. Given any two memory references it is necessary to determine whether

- they absolutely refer to the same memory location, or
- they possibly refer to the same memory location (cannot be determined at compile time),
- * or they absolutely do not refer to the same memory location.

The algorithm for the resolution of memory references is dependent on the host language. This algorithm is supplied by the host and encoded into CCG as conditional code (see HC\$IO, Section 9.3).

The design of the algorithm should take into consideration that it is executed on the order of N^2 times, where N is the number of variables involved.

[[PAGE BREAK-- end-21-20

Certain semantic information about a memory reference is basic to all languages & is required in the Internal Language load & store instructions (LD, ST, PLD, PST - Section 4.3.3) as follows:

IH the symbol table pointer which represents the base address of the reference.

RF specifies the variable offset from the base address
(cannot be determined at compile time)

CA specifies the constant offset from the base address.

The semantic information is sufficient for the resolution of references for FORTRAN. Other languages might require additional information in the symbol table.

The FORTRAN algorithm is summarized by the following four cases:

CASE	RESOLUTION
1. IH's are different	Absolutely NOT the same
2. IH's are the same RF'S are different	Possibly the same
3. IH's are the same RF's are the same CA's are different	Absolutely not the same
4. IH's are the same RF's are the same CA's are the same	Absolutely the same

Consider the references B(I+2)
 A(I*2)
 A(J+2)
 A(J+3)

Case 1 states that the reference to B is absolutely not the same as all of the references to A.

Case 2 states that the reference A(I+2) is possibly the same as all of the other references to A.

Case 3 states that the reference A(J+2) is absolutely not the same as A(J*3).

Case 4 states that A(J+3) is absolutely the same as A(J+3).

A simple, correct, but unrefined algorithm for PL/I introduces a storage class bit into the symbol table. This bit is set to 1 if the storage class of the variable is static, automatic, or controlled & set to 0 for all other storage classes.

If the AND of the storage class bit for two variables = 1 then the FORTRAN algorithm may be used. If it = 0, then it must be assumed that the references are possibly to the same Location.

[[] PAGE BREAK-- end-21-21

3.1.2 Straight Line Code Optimization

These optimizing transformations are performed unconditionally on all code, regardless of optimization level. The maximal unit of code over which each of these optimizations is performed is bounded by labels & unconditional Jumps (including Return Jump). They are not permitted to propagate across a label or an unconditional Jump.

REDUNDANT INSTRUCTION ELIMINATION

The process of redundant instruction elimination can be logically thought of in three steps.

1. Eliminate Redundant Memory References.

When two instructions are known to make reference to the same memory address and there is no intervening store instruction which references an address which is possibly the same, one of the memory references is eliminated as follows:

- Load-Load, the second load is eliminated
- Store-Store, the first store is eliminated (NOTE: This optimization is not propagated across a jump, conditional or unconditional.)
- Load-Store, the store is eliminated if the value being stored is the result of the load
- Store-Load, the load is eliminated

2. Eliminate Redundant Register Operations.

When two instructions have the same opcode & identical operands the second of the two instructions is eliminated.

The instructions being compared for redundancy are two operand instructions.

This fact limits the extent to which redundant expressions can be recognized & should influence the manner in which code is selected by the front end.

For example: $A+B$ and $B+A$ can be recognized as redundant, but $A+(B+C)$ and $(A+B)+C$ cannot.

Extraneous transmit instructions (BXI XJ) are considered redundant register operations and are eliminated.

3. Eliminate Unused Results

Operations which produce a result which is never used are eliminated.

For example, after load-store elimination, the statement $X=X$ is reduced to simply a load of X . The result is never used and is eliminated.

[[] PAGE BREAK-- end-21-22

CONSTANT EXPRESSION EVALUATION

When operands of an instruction are constants the instruction is replaced by an equivalent load of a constant. The evaluation is done on an instruction by instruction basis & will propagate through a sequence of instructions as long as all operands are constants.

A non-constant operand stops the constant evaluation.

Examples:

- | | Replaced by |
|------------------|---|
| a. $I = 3+4+2$ | $I = 7$ |
| b. $I = 3+4+J+1$ | $I = 7+J+1$ |
| c. $I = 3$ | $I = 3$ |
| $J = 4$ | $J = 4$ |
| $K = I+J$ | $K = 7$. After redundant loads are eliminated, $I+J$ is a constant |

ALGEBRAIC IDENTITIES

Expressions containing algebraic identities are reduced to simpler instructions as indicated in the following list. The letters in the list indicate:

I,J,K Integers X,Y Floating Point Z,W Indeterminate

& = LOGICAL AND, | = LOGICAL OR

Z & Z->Z
 Z & 0->0
 (-Z) & W -> BXI -XJ*XK INSTRUCTION
 0 | Z->Z
 Z | Z->Z
 XOR(Z,Z)->0
 XOR(0,Z)->Z
 0*I->0 1*I->I 2»I->I*I -1»I->-I
 I+0->I I-0->I 0- I->-I 1- I->0
 I+(-J)->I-J
 (J-K)+K->J
 (J+K)-J->K
 (J+K)-K->J
 (J-K)-J->-K
 J-(J+K)->-K
 J-(J-K)->K
 0*x->0
 0^x->0
 x*0->x
 x-0->x
 X-X->0
 1.»X->X
 2.*X->X+X
 -1.»X->-X
 X/1.->X
 X/z->-X ??
 X+X - NORMALIZE IS DELETED ??
 X*(-Y)->X-Y
 DBL(X-0)->0 ??
 DBL(0-X)->0 ??
 SHIFT(0,Z)->0
 SHIFT(Z,0!->Z
 SHIFT(Z,60!->Z,
 J*I*CON? A(J)->J=I+CON; ACI+CON)

[[] PAGE BREAK-- end-21-23

INSTRUCTION SCHEDULING

Code is scheduled for parallel operation to minimize execution time. Complicated code sequences are processed to minimize the number of temporaries required because of register set exhaustion.

7.1.3 Innermost Well-behaved Loop Optimizations

Innermost, well-behaved loop optimizations are performed when OPT<2 & they are specifically

requested by the Bridge (see CC\$OPTL, Section 8.4.1 for request method).

An innermost well-behaved loop contains only one entrance which must be at the top, contains only one exit which is either fall through at the bottom or a Jump to a label at the bottom, and does not contain any inner loops (including backward Jumps) nor return Jumps to external routines with side effects.

INVARIANT CODE MOTION

Subexpressions that are independent of the execution of the loop are moved out of the loop & evaluated in the prologue, if the expression is unconditionally executed or is conditionally executed but is incapable of causing an interrupt.

Invariant library function references are also moved (see FI. format, word R2, Section 4.5).

Example:

```
DO 10 1*1,N
A(1) = X*Y • D(J)      MOVED
B(I) = SIN(Y+Z)         EVAL OF SIN MOVED
IF( A(I) .NE. 0) GO TO 10
C(I) = X/Y              NOT MOVED, Y COULD BE 0
10  CONTINUE
```

[[] PAGE BREAK-- end-21-24

STRENGTH REDUCTION OF INTEGER POLYNOMIALS

For the purpose of this document, an integer polynomial is defined to be an expression which is a linear function of an iteratively defined variable & which is a candidate for a B register.

A linear function of I is:

$k_1 \cdot I + k_2$ where k_1 & k_2 are invariant (during the loop) expressions.

An iteratively defined variable is:

$I = I + k_3$ where k_3 is a constant or an invariant (during the loop) variable.

An integer polynomial is reduced to remove the integer multiplies and replace them by adds. The initial value of the polynomial is $k_1 \cdot I + k_2$ and each time through the loop it is incremented by $k_1 \cdot k_3$.

LOOP REGISTER-ASSIGNMENT & Prefetch Timing example

Addresses, short constants and integer polynomials are assigned to B registers.

Scalar variables that are not in LCM may be assigned to X registers in innermost loops.

Array references in small loops may be "prefetched- if they are unconditionally executed, the subscript is an integer polynomial and the increment is small.

"Prefetching- is a speed optimization which permits execution in parallel of the array element fetch with the increment, test and jump.

Example:

```
S=0
DO 10 I = 1,1000
10  S = S + A(I)
```

	TRADITIONAL LOOP CODE	cycles - critical path
	SB1 1 ; I = 1	
	SB2 A-1	
	MX6 0 ; S = 0	
	SB4 A 4-999	
L10	SAI B1*82 ; X1 = A(I)	8
	SB2 82 + B1 ; I = I + 1	executes before Fat add can execute
	FX5 X6 + X1	3 - execution waits on memory fetch
	NX6 X5 ; NORMALIZE THE FLOATING ADD	3 - jump back can execute in next cycle
	LT B2,84,L10 ; LOOP BACK	3 for a total of 17
	SA6 S ; STORE RESULT	

[[] PAGE BREAK-- end-21-25

	Wait for fetch occurs, Options "OPT=2,U0"
	Prefetching Loop
	SB1 1 ; I = 1
	SB2 -999
	MX6 0 ; S = 0
	SA1 A ; PRE FETCH A(1)
L10	FX5 X6 + X1 ; 3 cycles
	SAI A1+B1 ; PRE FETCH (8 cycles) executes in parallel with next 3 instructions
	SB2 B2+B1 3 but only 2 on critical path
	NX6 X5 ; 3 but not on critical path
	LT B2,L10 ; back to L10 at cycle 9 or 10
	SA6 S

3.1.4 Global Optimizations

The optimizing transformations discussed in Sections 3.1.2 and 3.1.3 are accomplished strictly by looking at one code sequence at a time. The global optimizing transformations which are performed when OPT=2 make use of two additional data structures which permit analysis of the entire program: control flow information and use/definition information.

The Bridge supplies sufficient information for the Code Transformer to analyze the program flow. It does so by dividing the source program into code sequences which represent the nodes of the control flow graph and providing a table whose entries represent the edges of the graph (Sections 4.2 and 4.4).

The Code Transformer, aided by the Bridge (Section 4.5) constructs a table which records the uses (loads) and definitions (stores) of every memory location or class of memory locations referenced in the program. The information is extracted from the instruction sequences and from information about the actual parameter lists of procedure references supplied by the Bridge.

Each memory reference has a use/definition class u(i), associated with it as follows:

- The reference may address a set of locations, indexed by a subscript function. The reference A(I) may reference any of the locations A(1), A(2), ..., A(N) and is called a class reference
- The reference may only address one location. Examples are single precision scalar variable X, which is called an element reference, and A(2) which is called a class member of class A. Multiple precision scalars generate n element references. For example, a reference to a double precision variable, D, generates the use/definition classes for the locations D+0 7 D+1.

Let $U(1)$, $U(2)$ be two use/def classes. Present CCG algorithms assume that if the intersection of $U(1)$ and $U(2)$ is non-empty, then one is a subset of the other, and if they are not identical then the smaller set has only one element. This assumption is not valid for PL/I nor other languages whose storage classes are more general than those of FORTRAN. The Global Optimizer cannot accommodate such languages until these algorithms are changed.

[[] PAGE BREAK-- end-21-26

The control flow information permits the Global Optimizer to detect every single entry loop. Loop optimizations (discussed in Section 3.1.3) are then performed on every loop starting with the innermost loops and working outwards. Invariant code, for example, might be moved from an innermost loop clear out of a nest of loops.

The use/definition information permits the Global Optimizer to determine for every reference in a code sequence whether it is used, defined, used before definition, and/or is live on exit (used in a subsequent sequence without prior definition). This permits detection of definitions which are not subsequently used and, hence, may be eliminated together with their associated computations. It also permits delaying eligible stores to the proper exit path from a loop.

An additional optimization, linear function test replacement, is accomplished utilizing the control flow information and use/def information. If, after register assignment in a loop, the loop control variable has no uses other than in the increment and test, then it will be eliminated and replaced by a linear function. In order for this optimization to take place, all references to the control variable, the increment, and the limit must have the D.RF bit set in the instruction descriptor word (see Section 4.3.5). This bit indicates the variables are B-register candidates.

```

      REAL A(10,10)
      DO 10 I=1,10          DO 10 I1 * 7,97,10
10  A(7,I) * X          10  A(I1) = X

```

The Global Optimizer performs more extensive register assignment algorithms than are performed for OPT<2. Subscript calculations are assigned to B-registers in long sequences of straight line code to free up X-registers for use by 60 bit quantities.

3.2 AUXILIARY FUNCTIONS

3.2.1 Code Transferor Generated Temporaries

During the scheduling process and during some optimizing transformations, the Code Transformer generates instructions which require temporary storage. Two methods of temporary storage management are provided: static storage and stack storage. The method used is selected by HC\$ID (Section 9.3).

[[] PAGE BREAK-- end-21-27

STATIC STORAGE

The predefined symbol OT. (Section 2.1.4) is used as the base address of the temporaries generated by the scheduler and IT. is used as the base for optimizer generated temporaries. The Code Transformer manages temporaries to provide maximum overlap of storage use.

It will be seen later that the End Processor is responsible for reserving storage for the two temporary arrays.

STACK STORAGE (used by PL/I)

A reference to stack storage is defined to be to AQ+offset.

The Bridge directs the Code Transformer in its use of offsets by providing an abbreviated symbol table, A0 and an ordinal into the table.

The A0 table contains a one word entry for each stack frame (PL/I activation descriptor) whose content changes during code generation as follows

The Bridge supplies the table as: 42/0,18/initial_Length

The Code Transformer changes it to: 24/0,18/temporary_length,18/initial_Length

The End Processor changes it to: 42/0,18/final_length (initial plus temporaries)

Each time the Bridge calls the Code Transformer it indicates which stack frame is in effect by placing the appropriate AD ordinal in H0\$PN. The Code Transformer makes reference to temporaries by AQ+initial length+i where i is a running counter of the temporaries in effect for the code sequence. It records the maximum i in the temporary length field. If code is required to reference the length of the stack frame (e.g., calling sequences for runtime storage management) the reference is made by using an IH which indicates the appropriate entry in the A0 table.

3.2.2 Address Expansion and Substitution

The Code Transformer provides for expanding references to certain types of variables & provides a mechanism for doing substitution of addresses at runtime. These mechanisms are host compiler dependent and are encoded into CCG as conditional code (conditioned by HC\$I0, Section 9.31.

They will be fully implemented for any language upon receipt of a specification.

If no address expansion is to take place, the AET field in the symbol table (Section 2.1.1.2) must not be defined. If address substitution is not used, HC\$FPAS (Section 9.3) must be set to 0.

FORTTRAN ADDRESS EXPANSION

The FORTRAN AET field consists of the LCM & FP bits in symbol table Word B (Section 2.1.1.2). A ref to a var, whose LCM bit is set, is changed to the appropriate LCM access instruction. A ref to a var, whose FP bit is set, is modified to indirectly address the variable via A0 and the APIist. The position in the APIist is specified by the FPO field in Word B.

[[] PAGE BREAK-- end-21-28

FOPTRAN ADDRESS SUBSTITUTION

CCG determines which formal parameter references are referenced directly (subbed), keeps track of the number of -SUBS- for each parameter in a host supplied table (FPI), & at assembly time outputs the required code FOR the SUB macros.

AH level 0 F.P. references are subbed. The count of the number of level 0 references for a given F.P. are also kept in FPI.

3.2.3 Address Assignment for Labels

Each time the Code Transformer encounters a LAB pseudo operation (Section 4.3.3), It records its block relative address in Word C of the symbol table. The End Processor must change all block relative addresses to program relocatable addresses (discussed in Section 5.2.2).

3.2.4 Code Length Computation

The Code Transformer increments block lengths for those local blocks that it processes. When it is initialized (by a call to CG\$INIT) the Code Transformer is informed of the local block that is currently in effect on the SLIST file. During initialization the Code Transformer initializes the entry points CC\$LB0, CC\$BLEN & CC\$PC. After initialization, but prior to processing any code sequences, the Bridge may call the routine CG\$CUB (Section 8.4.2) to change the local block. This routine updates LBT, reinitializes the above-mentioned entry points and writes a USE declarative to the SLIST file. During the Bridge/Code Transformer operation, the local block may not be changed. It will be seen in Chapter 5 that code may be issued to different blocks during end processing. At the end of code transformation LBT reflects the lengths of all blocks except the current one. The current block length is reflected in CC\$BLEN and CC\$PC. LBT has not been updated. This is left to the End Processor and is discussed in Chapter 5.

[[] PAGE BREAK-- end-21-29

3.2.5 Materialization of Variables and Constants

The Code Transformer provides a record keeping service which permits the host compiler to eliminate storage reservation for variables and constants which are not used. After it has performed all of its optimizing transformations it sets the MAT bit (Section 2.1.1.2) for each variable which is referenced and the CUT entry (Section 2.2.2) for each constant which is referenced.

The host compiler must keep track of variable and constant references that the Code Transformer does not see (e.g., actual parameter lists).

3.2.6 Dead Code Error Messages

CCG does not write to the output file.

When OPT=2 detects dead code the host routine "HR\$LOC" will be called with the table "RND" containing the list of line numbers (in binary) which are unreachable. The host should put out an error message similar to the following - "Statements beginning at the below line numbers are unreachable (dead code) & will not be processed" followed by the list of line numbers. On returning to CCG the table area will be released.

[[] PAGE BREAK-- end-21-30

4.0 THE BRIDGE

The Bridge is the first routine to gain control in the Code Generator overlay and as such must initialize it. After initialization the primary function of the Bridge is to segment the intermediate text into code sequences, translate them into the Internal Language of the Code Transformer and present them to the Code Transformer one at a time. If OPT=2 the Bridge must also collect control flow information and use information.

If the Front End has not done so the Bridge must bring the symbol table and associated tables to

the required state as described in Chapter 2.

It must also assure that text which does not generate machine instructions is written to the file SLIST as is described later in Chapter 6.

The general flow of the Bridge is as outlined in the steps below.

- a. If the Front End has not done so, bring all tables to the state required by the Code Transformer, as discussed in Chapter 2.
- b. Initialize the table vectors and move tables as required to bring them to the state described in Section 8.3.
- c. Initialize CC\$SRF (Section 8.4.1) to indicate if address expansion is required.
- d. Call CG\$INIT (Section 8.4.1) to initialize the Code Transformer and inform it of the local block which is in effect on the SLIST file.
- e. If desired, call CG\$CUB (Section 8.4.2) to specify a new local block into which all code processed by the Code Transformer will be placed.
- f. Collect a code sequence (Section 4.2) and translate it into the Internal Language (Section 4.3). If OPT=2, collect control flow information (Section 4.4k) and use information (Section 4.5).
- g. If the code sequence is an optimizable loop (Section 4.2) set CC\$OPTL (Section 8.4.1) to indicate.
- h. If stack storage has been selected for temporaries (Section 3.2.1), set HO\$PN to indicate the current stack frame.
- i. Call CG\$PAS which is the main entry point of the Code Transformer.
- j. If the entire source program has not been processed, go to f. k. Exit.

[[] PAGE BREAK-- end-21-31

CODE SEQUENCES

The Bridge segments the program into code sequences and presents them to the Code Transformer one at a time.

The selection of a Code sequence is influenced primarily by the level of optimization but must also be constrained by core requirements. Each instruction in the code sequence requires four words to represent it and during the scheduling process an additional average of four words per instruction is required.

The various alternatives for code sequences are discussed below*

SINGLE SOURCE STATEMENT

A single source statement is normally the minimal code sequence that is presented to the Code Transformer and is usually used for the lowest optimization level (OPT=0). A statement can be broken into segments and the segments used as code sequences as long as it is broken at boundary markers (Jumps and labels) and follows the restriction (that will be seen later in the discussion of the Internal Language) that every operand in a sequence must be defined as the result of a preceding instruction in the same sequence.

EXTENDED BASIC BLOCK

It has the property that if a statement is executed then so have all of its predecessors.

An extended basic block may begin with a label and may end with an unconditional Jump, but it may not contain any additional labels or unconditional Jumps.

As was discussed in Section 3.1.2, the straight line optimizations do not take advantage of a sequence which includes more than an extended basic block. Hence, a sequence which is as close as possible to an extended basic block (but does not break in the middle of a statement) is optimal for the straight line optimizations and is usually used for OPT*1.

INNERMOST WELL BEHAVED LOOP

An innermost, well-behaved loop contains only one entrance which must be at the top, contains only one exit which is either fall through at the bottom or a Jump to a label at the bottom, and does not contain any inner loops (including backward Jumps) nor return Jumps to external routines with side effects.

When such a loop is encountered, it is used as a code sequence if the innermost loop optimizations discussed in Section 3.1.3 are desired (generally OPT*1).

[[] PAGE BREAK-- end-21-32

BASIC BLOCK

It has the property that if one instruction in the block is executed, all are executed.

A basic block may begin with a label and may end with a Jump, but may not contain any additional labels or Jumps (conditional or unconditional). It may contain a return Jump if the return is guaranteed.

A basic block is the code sequence that must be used for OPT=2.

4.3 THE INTERNAL LANGUAGE

The internal language (IL) of the Code Transformer consists of a set of elementary instructions, most of which correspond to machine instructions, some of which are pseudo instructions to direct the code transformation process.

The IL is a low level language where all operations are explicit to the machine instruction level. Each IL instruction contains an opcode identifying the instruction and information fields which may be:

- * operand links (R-numbers) between 2 instructions
- * symbol table pointers
- * constants
- * machine register ordinals
- * miscellaneous information (pseudo instructions only).

4.3.1 Operand Links (R-Numbers)

An understanding of the concept of operand links (R-numbers) is basic to the understanding of the IL. Every machine instruction which produces a result has associated with it an R-number which is used as a label (or link) in succeeding instructions which use the result.

For example:

A * B * C is represented in IL as:

```
LD R10,,,B    "Load B (R10 is arbitrary for this example)"
LD R11,,,C    "Load C"
FM R12,R10,R11    "B»C - Result R12 = R10 * R11"
ST R12,,,A    "Store Result into A"
```

When a machine instruction is finally assembled, the R-numbers result in X-registers unless they are specifically forced to B-registers or A0 (via the RS pseudo instruction discussed below).

R-numbers never represent registers A1 through A7. That is to say, the IL does not permit explicit manipulation of A1 to A7. Control of these registers is implicit to the code transformation process. As was stated earlier, the IL is explicit to the machine instruction level. Instructions are not implicitly inserted by the Code Transformer. This imposes the restriction that the uses of an R-number be constrained to a compatible register set.

For example, the result of a load instruction cannot be used as the source of a store instruction.

The load results (X1 to X5) are incompatible with the store requirements (X6 to X7).
A transmit instruction must intervene. On the other hand the result of a floating point operation can be used as the source of a store since it is possible for the result to be in a compatible register (X6, X7).

[[] PAGE BREAK-- end-21-33

Conventions which must be observed in the use of R-numbers are:

a. Definition before use

An R-number must be defined in a sequence before it is used as an operand.
Exceptions are: R-number 0 represents B0 (or no operand) & R-number 1 represents A0.

b. Uniqueness

No R-number may define more than one result in a sequence.

c. Range

R-numbers must be contained within ranges according to one of the two methods described below.
The reason for the restrictions is that the code transformer converts the R-numbers into "canonical" form where they are indices into the sequence they are in (multiples of 4).
The choice of method is defined by the value of HC\$ROL (see Section 9.3).

1. This method assumes that the Front End assigns a large range of R-numbers over the entire source program (range 1) and that the Bridge superimposes a separate range of R-numbers for each code sequence (range 2).

By convention, the Front End starts assignment at the beginning of the program with R-number 4 and continues to 77777B. (32,767 DIV 4 = 8,191 IL instructions is max number in a sequence)
The Bridge starts assignment at the beginning of each code sequence with 100002B & continues to 177777B.
Note that for a very large program, the Front End R-numbers may need to start over.
The Front End and Bridge must have some protocol to cause the start over to happen at the beginning of a code sequence.
It may not happen in the middle of a sequence because this would produce a range of R-numbers which is very large.

The Code Transformer uses an n-word table, where n is the sum of the size of the two ranges for the sequence. For this reason also, the R-number ranges should be kept as compact as possible. When a code sequence is passed to the Code Transformer, the two arrays CC\$BRN and CC\$BIR (see Section S.k.1) must contain:

CC\$BRN(1) = Smallest range PASS 1 R-number in the sequence.
CC\$BRN(2) = Largest range PASS 1 R-number in the sequence plus one.
CC\$BIR(1) = 100002B.
CC\$BIR(2) = Largest range PASS 2 R-number in the sequence

After each call to CG\$PAS, CC\$BRN(1) is set equal to CC\$BRN(2) & CC\$BIR(2) = 100002B

2. This method assumes that only the Bridge assigns R-numbers and that it starts over with R-number at the beginning of each sequence.

When a code sequence is passed to the Code Transformer the array CC\$BRN must contain:

CC\$BRN(1) = 4
CC\$BRN(2) = Largest R-number in the sequence
After each call to CG\$PAS, CC\$BRN(2) = 4

[[] PAGE BREAK-- end-21-34

4.3.2 Instruction Types

IL instructions are divided into four types, each having its own distinct format.
The types are:

TYPE	Format	Description
I	OPC RI,RJ,RK	Two and three address instructions such as BXI XJ & FXI XJ»XK.
II	OPC, RI,IN,S0	One address instructions such as SXI con & MXI con.
III	OPC RI,RF,CA,IH,H2	Two address instructions, memory refs & conditional Jumps such as NZ XI,symbol or SAI RF+CA+symbol •
IV	OPC CA,IH	Some pseudo instructions & the unconditional Jumps such as EQ LAB.

The mnemonics in the above formats are used to represent the following items:

MNEMONIC	Item
OPC	Operation code identifying the instruction.
RI,RJ,RK,RF	R-numbers
	RI represents the result register for instructions which produce a result. For stores and conditional Jumps, RI is an operand register. Pseudo instructions vary in their use of RI. RJ & RK represents an operand registers
IN,CA	RF is the register containing the variable portion of an address calculation. 18 bit constants. IN used to represent constants that are in instructions such as the mask. CA the constant portion of an address calc or the index into an aux table
S0	Used to specify a particular register. This field exists in all Type II instructions but is meaningful only in the RS and OEF pseudo instructions. Its description is contained with the definition of the RS instruction.
H2,IH	Symbol table pointers. H2 contains an ordinal only & impoliticly references SYM (may not reference an abbreviated symbol table).

[[] PAGE BREAK-- end-21-35

4.3.3 IL Instruction Set

Each instruction of the IL is discussed in this section.

PSEUDO INSTRUCTIONS

The pseudo instructions are used to direct the code transformation process.
They do not result in the generation of a machine instruction.

TYPE	OPCODE	DESCRIPTION
------	--------	-------------

IV	BOS	Beginning of Statement
----	-----	------------------------

This instruction must be the first instruction in a sequence. BOS instructions which appear in the middle of a sequence are eliminated by the Code Transformer.
The format of this instruction does not conform to the format of its type so it is presented here:

12/PCBOS),18/statement number,2/list flags,10/0,18/available.

When the object listing is selected the BOS causes the generation of a comment line with the statement number in it. The list flag field is used to toggle the object listing. Its values may be

0 no change
1 object list on
2 object list off

IV	EOS	End of Sequence.
----	-----	------------------

Used to artificially stop code motion during the scheduling process.
is guaranteed to not drift in either direction across an EOS.
CA & IH must both = 0

Code

[[PAGE BREAK-- end-21-36

II	RS	Register Specify.
----	----	-------------------

Used to specify that the result of the immediately preceding instruction is to be assigned to a specific register.

The RI of the RS instruction must be the same as the RI of the immediately preceding instruction with the following exception. If the preceding instruction is an Unpack (UP) or Normalize (NR) instruction with a non-zero RJ (two results) the

RS instruction serves as a specification of this RJ and is called an RJRS.

In this case, the RI of the RS must be the same as the RJ of the UP or NR instruction.

The IN field of the RS is set to 1 for an RJRS and set to 0 otherwise.

The S0 field is used to specify the register and provides lock information about the register.

The format of the S0 field is: 6/reserved,2/lock type,3/reg type, 3/reg number

The register type field takes on the values of 0, 1, 2 for B, A, and X respectively.

The register number field takes on the values 0 through 7.

Please note that registers A1 - A7 are not permissible.

The lock type field has the following meaning!

0 This type of lock is used to specify that the register content must remain unchanged until the next Return Jump, Unconditional Jump, or Indexed Jump is issued.

It is intended for use to lock an argument in a register until the call is made, even though the argument has no uses which are apparent to the Code Transformer.

1 This type of lock permits the content of the specified register to be released when there are no more uses of it.

2 Reserved for the Code Transformer.

3 This value must appear if the RS is an RJRS.

[PAGE BREAK-- end-21-37

Some examples of the use of the RS instruction are:

a. Assume that an argument is to be passed in X1. The statement CALL PROC(A) looks like:

```
LD R10,,,A
RS R10,0,X1 lock type = 0
RJ PROC OELIMITER
```

b. For this example, please look ahead to the Memory Reference & Set instructions & note that a number of them have a choice of A, B, or X registers for the I-register. This register will always result in an X-register unless the RS instruction is used to force it to a B-register or A0.

Note also that a number of them have a choice of registers for the J-register.

Selector of this register is determined by the RI of the instruction that the RJ links to.

This example using a Set (S) & a Short Add (SA) instructions illustrates the effect of the RS.

IL Instructions	Resulting Machine Instructions
S R10,5	SB2 5
RS R10,0,B2	
S R11,6	SXn 6 "XN,available X-register
SA R12,R11,R10	SB4 Xn+B2
RS Ri2,0,B4	

I This example illustrates the use of an RJRS.

```
UP R11,R12,R10 Unpack R10 into two results, R11 goes into Xn
RS R12,1,B2 This instruction must appear to specify the R12 result.
R11 cannot be specified. The lock type field must contain 3
```

II DEF Register Define.

DEF is used to define an R-number and to specify that it is assigned to a specific register.

It is used, for example, to specify a value returned by a function call.

A DEF may only occur at the beginning of a sequence or after a DEF or a boundary marker (jumps and labels). The IN field is always 0 and the SO field is as specified in the RS instruction except that the lock type is always zero.

IV LAB Label Definition.

LAB defines a label (IH BSS CA) and reserves storage.

[PAGE BREAK-- end-21-38

COMPUTATIONAL INSTRUCTIONS

IL instructions in this group have a one-to-one correspondence with machine instructions

TYPE	IL	Machine	NAM
	XMT	RI,RJ BXI XJ	Transmit
I	AND	RI,RJ,RK RXI XJ*XK	And
I	OR	RI,RJ,RK BXI XJ+XK	Or
I	XOR	RI,RJ,RK BXI XJ-XK	Exclusive or

I	XMTC	RI,RJ	BXI -XJ	Transmit complement
I	STR	RI,RJ,RK	BXI -XJ*XK	Stroke
I	IMP	RI,RJ,RK	BXI -XJ+XK	Implication NAND
I	EQV	RI,RJ,RK	BXI -XJ-XK	Equivalence NOR
III	KLS	ri,rf,ca	LXI JK	Constant left shift
III	KRS	ri,rf,ca	AXI JK	Constant right shift
I	ILS	RI,RJ,RK	LXI BJ,XK	Indexed left shift
I	IRS	RI,RJ,RK	AXI BJ,XK	Indexed right shift
I	NR	RI,RJ,RK	NXI BJ,XK	Normalize
I	RNZ	RI,RJ,RK	k ZXI BJ,XK	Rounded normalize
I	UP	RI,RJ,RK	UXI Bj,XK	Unpack
I	PK	ri,rj,RK	PXI Bj,XK	Pack
I	FA	ri,rj,rk	FXIC XJ+XK	Floating add
I	FS	ri,rj,rk	FXI XJ-XK	Floating subtract
I	DFA	ri,rj,rk	DXI XJ+XK	Double floating add
I	DFS	ri,rj,rk	DXI XJ-XK	Double floating subtract
I	RFA	ri,rj,rk	RXI XJ+XK	Rounded floating add
I	RFS	ri,rj,RK	RXI XJ-XK	Rounded floating subtract
I	IA	ri,rj,rk	IXI XJ+XK	Integer add
I	IS	ri,rj,rk	IXI XJ-XK	Integer subtract
I	FM	ri,rj,rk	FXI XJ*XK	Floating multiply
I	RFM	ri,rj,rk	RXI XJ*XK	Rounded floating multiply
I	DFM	ri,rj,rk	DXI XJ»XK	Double floating multiply
II	FMA	ri,ca	MXI JK	Mask
I	FD	ri,rj,rk	FXI XJ/XK	Floating divide
I	RFD	ri,rj,RK	RXI XJ/XK	Rounded floating divide
I	CX	ri,rj	CXI XJ	Count bits

[[] PAGE BREAK-- end-21-39

MEMORY REFERENCE.AND SET INSTRUCTIONS

Each IL instruction in this group results in one machine instruction selected from a specific set. The notation "ZJ" is used to denote a choice of registers from the set Bi, Xi or A0 and, hence, a set of instructions.

The selection of the particular Machine instruction is dependent on the context in which the IL instruction is used (as explained in the definition of the RS instruction).

Within the machine instruction specification IH & H2 are used to represent the symbols that corresponding IL IH & H2 point to.

TYPE	IL INSTRUCTION	MACHINE	NAME
III	LD RI,RF,CA,IH	SAI AI+CA+IH	LOAD
III	LD RI,RF,CA,IH	SAI AI+CA+IH	STORE
III	STT RI,RF,CA,IH,H2	SZI ZJ+CA+IH+H2	STORE TRANSMIT

NOTE: If H2 is non-zero then IH must be non-zero (negative relocation is not permitted).
H2 may only point to the main symbol table.

I	PLD	RI,RJ,RF,CA,IH	SAI ZJ+CA	Parametric Load
I	PST	RI,RJ,RF,CA,IH	SAI ZJ+CA	Parametric Store

NOTE: RI & IH DO NOT enter into the machine instruction.
they are used to determine the class of memory locations referenced.

II	S RI,IN	SZI IN	SET
I	SA RI,RJ,RK	SZI ZJ+BK	SHORT ADD
I	SS RI,RJ,RK	SZI BJ/AO-BK	SHORT SUBTRACT

JUMP INSTRUCTIONS

The low order 6 bits of the CA field are used to hold the Jump type for the conditional jump instructions (JPX, JPBB, RJXJ). The Jump types are defined in CMPLTXT (Section 9.2.1) by the symbols JC.XX (XX = ZR,NZ,PL,MI,OR,IR,DF,ID,EQ,NE,GE,LT).

[PAGE BREAK-- end-21-40

TYPE	IL INSTRUCTION	MACHINE	NAME
III	JPX RI,0,JT,IH	JT XI,IH	Conditional X Jumps

III	JPBB RI,RF,JT,IH	JT BI,BF,IH	Conditional B Jumps
-----	------------------	-------------	---------------------

III	RJXJ RI,RF,12/CA, 6/JT,IH	7JT,RI,*+1, RJ IH+CA	Conditional Return jump
-----	------------------------------	-------------------------	-------------------------

NOTE: 7JT is the reverse test of JT (7ZR = NZ). RF is the R-Number of result that must not be used until the RJXJ is executed.
This is because the RJXJ is not considered a boundary marker, code may float to either side of it. Uses of RF are prevented from floating in front of the RJXJ).

TYPE	IL INSTRUCTION	MACHINE	NAME
III	JIN RI,0,0,IH	JP BI+IH	indexed jump
IV	RJ3 0,IH	RJ IH	30 BIT Return Jump
IV	RJ6 CA,IH	+RJ IH VFD 12/CA.18/HC\$RJTBN	60 BIT Return Jump

NOTE: See Section 9.3 for definition of HC\$RJTBN.

IV	UJP 0,IH	EQ IH	Unconditional Jump
----	----------	-------	--------------------

SPECIAL CASE INSTRUCTIONS

This group have been created to take care of various special cases

TYPE	IL INSTRUCTION	MACHINE	NAME
I	IAZ RI,RJ,RJ	IXI XJ+XK	Integer Add Zero
I	IAS RI,RJ,RK	IXI XJ-XK	Integer Subtract Zero

In the above two instructions, RJ must be the result of a CLR instruction. These instructions are used when it is necessary to inhibit the reduction of the algebraic identities $0+X$ and $0-X$. The add/subtract is not optimized out.

I	IM RI,RJ,RK	DX1 XJ»XK	Integer Multiply
---	-------------	-----------	------------------

It is assumed that this instruction will not cause an interrupt and hence, may be moved from conditionally executed code. The operands must be such that an interrupt will not occur (e.g. subscript calculation).

II CLR RI,0 MXI 0 or BX1 XI-XI Clear a Register

[[PAGE BREAK-- end-21-41

I SXT RI,RJ BXI XJ Shift Transmit
This instruction must be the immediate predecessor of each KLS/KR* instruction •

III LDC RI,0,CA,IH SAI CA+IH Load Constant

CA is an index into CVT (Section 2.2.2) & IH is the symbol table ordinal of "CON." (Section 2.1.4). This instruction permits the Code Transformer to evaluate constant expression: and to compact the constant table when unused constants are detected. I must be used for all references to constants in CVT.

III LDV RI,0,CA,IH SAI CA+IH Load Vardim

CA is an index into VDI (Section 2.2.3) & IH is the symbol table ordinal. This instruction allows the host to minimize the number of temporaries needed to setup variable dimension initialization code by deferring the decision until end processing when it is known which one were actually used.

4.3.4 Instruction Set Rules

This section contains rules that apply to certain instructions and provides a few suggestions for the use of the instructions.

1. The KLS/KRS instructions must be immediately preceded by an SXT instruction (the RF of the KLS/KRS must be the RI of the SXT). Also the KLS/KRS instructions may not be used as the source of a store instruction unless a transmit intervenes.
2. The SXT instruction may only be used as a constant shift predecessor.
3. Whenever a 0 in an X-register is needed, the CLR instruction must be used.
4. Do not use the STT instruction when IH and CA = 0; use an SA instruction instead.
5. Do not use an LD for reference to values in CVT. An LDC must be used for these values.

4.3.5 Instruction Formats

When CG\$PAS is called, a code sequence is presented to it in the table TXT. Each IL instruction occupies 4 contiguous words of memory. The words are called the RI word, the R2 or IH word, the descriptor word & the link word.

[[PAGE BREAK-- end-21-42

R1 WORD

The format of the RI word varies from type to type, but the upper 12 bits always contain the IL Instruction opcode with a bias of 2080B due to the Pack instruction (referred to as P(OC)) . The format of the word for the different types is*

```

I      12/P(OC),16/RJ,16/RK,16'RI
II     12/P(OC),18/IN,14/S0,16/RI
III    12/P(OC),18/IN,12H2,2/0,16/RI
IV     12/P(OC),18/CA,12/0,18/IH

```

Field definition macros for describing these formats are:

```

OESCRIBE R1.
OC      OEFINE 12
RJ      OEFINE 16
RK      DEFINE 16
RI      OEFINE 16

      REOEF RJ
IN      OEFINE 18
S0      OEFINE 14

      REOEF RJ
CA      OEFINE 18
H2      OEFINE 12
IH      OEFINE 18

```

R2 OR IH WORD

The primary function of the R2 word is to hold the semantic information required for memory references (Section 3.1.1) Its format for all Type III and all memory reference instructions (Type III,PLD, PST) is:

```

6/0,18/RF,18/CA,18/IH

```

Field definition macros for describing this format are:

```

DESCRIBE IH.
CCG     DEFINE 6
RF      OEFINE 18
CA      OEFINE 18
I       OEFINE 3
H       OEFINE 15
IH      OEQU  H,18

```

If OPT=2, the format of the R2 word for the RJ3/RJ6 instructions is the "FI." discussed in Section 4.5. For all other instructions the R2 word must be zero.

[[] PAGE BREAK-- end-21-43

DESCRIPTOR WORD

The descriptor word contains property bits which are statically associated with the opcode. These property bits are provided in the table F\$RDT, access to which is described in Section 8.4.2.

The only other bit in this word that the Bridge is concerned with is D.RF. In order for the linear function test replacement optimization (Section 3.1.4) to take place, reference to variables that are B-register candidates must have this bit set.

LINK WORD

This word is used by the Code Transformer for various purposes & must be initialized to zero.

4.4 CONTROL FLOW INFORMATION

When OPT-2, the code sequences that the Bridge presents to the Code Transformer are basic blocks (Section 4.2). The salient feature about a basic block is that there is no transfer of control into or out of the middle of the sequence. It is entered at the top, and once entered, the entire sequence is executed. The exit at the bottom may be a fall through to the next block, a Jump to another block or

(in the case of a conditional Jump) both a fall through and a jump.

The Bridge must record all transfers of control from one block to another in the table CFT.

The format of a CFT entry is CF. format which is 1/JP,29/FROM,30/T0.

The basic blocks are numbered sequentially (starting with 2) in the order they are presented to the Code Transformer.

FROM is the block number of the block from which control is transferred.

If JP=0, T0 is the block number to which control is transferred.

If JP=1, T0 is a symbol table pointer indicating the label to which control is transferred.

Block number 0 is the pseudo exit block and block number 1 is the pseudo entry block.

CFT must contain an entry (k,0) for each block k, which exits the subprogram and an entry (l,k) for each block which is an entry point.

The cell CC\$CBN holds the current block number: the Bridge is responsible for incrementing it at the beginning of each new basic block. When CG\$PAS is called to process a basic block, CC\$CBN must hold the block number of the block in TXT. CG\$INIT initializes CC\$CBN to 2 and adds the entry (1,2) to CFT. Whenever a transfer label definition is encountered, the Bridge must call CG\$LABD (X1=IH) to define the block number associated with the symbol.

4.5 USE/DEF INFORMATION

When OPT*OPT=22, the Bridge must aid the Code Transformer in collecting use/def information (Section 3.1.4) by supplying information about the actual parameter lists for procedure calls. The protocol for collecting this information for each Return Jump instruction is:

[[PAGE BREAK-- end-21-44

1. Create a list of the variables referenced by the procedure (actual oarameters) in a table in "AP." format which is:

```
      DESCRIBE AP. OEFINE 1
ID      DEFINE 1      =1 Is a param list for a Library read routine

USE      DEFINE 1      =1 is a read param, but input may not occur
              (FORTRAN NAMELIST, PL/I data directed GET

P1      DEFINE 1      =1 for a double or Complex Variable (occupies 2 words)

      DEFINE 2
```

CR DEFINE 1 =1 if a Class Ref (Has a variable subscript)

DEFINE 18

CA DEFINE 18 Constant offset from base address

IH DEFINE 18 Symbol table ordinal

2. CALL CG\$CPL(X6sTBLP, X5sINØEX, B2=LEN) where

T8LP is the address of the table which contains the list.

INDEX is the index into the table for the start of the list.

LEN is the number of entries in the list.

CG\$CPL returns

XI =index to the list in the table IOL

XØ =Length of the list it added to IOL

X7 =L\$IØL

3. Create the R2 word of the RJ3 or RJ6 instruction in FI. format which is:

DESCRIBE FI.

FI OEFINE 29 Function Type
 1 - user function
 2 - basic external function (library does not modify any variables)
 3 - I/O function

REGP OEFINE 24 Bit mask of registers preserved - basic external func only
 (Ø-BØ, 1-B1»...23-X7)

INOX OEFINE 18 Index to list in IOL

LEN DEFINE 12 Length of list in IOL. Either Ø or 1S1Ø plus length.

[[] PAGE BREAK-- end-21-45

4.6 Packing of ST.'s (Function Temporaries)

In FTN 5 the ordinal (CA) of the memory references to the elements of the ST. array cannot be determined prior to calling CG\$PAS. To get around this problem, the bridge assigns tentative values to the CA's. After the code transform is finished eliminating redundant references, it calls BR\$AFT (in the bridge) to scan the sequence in TXT & adjust the CA's of the ST. references.

[[] PAGE BREAK-- end-21-46

5.Ø THE END PROCESSOR

The End Processor is responsible for completing the SLIST file and for bringing all tables to the state required by the Assembler. The specific functions that it must perform are discussed in the following sections. Other functions may be performed at the host's discretion.

For example, if the host wishes to take advantage of the MAT bit (Section 2.1.1.2) it would delay storage reservation for all variables until end processing.

End processing may be a convenient time for producing a reference map.

5.1 COMPLETING THE SLIST FILE

The End Processor may wish to issue prologue code. It must output constants, reserve storage for temporaries and write an END pseudo on the SLIST file.

5.1.1 Prologue Code

If desired, the End Processor may call on the Code Transformer to process code sequences. These sequences are collected and presented to the Code Transformer identically as they were during Bridge processing except that CG\$CPC is called instead of CG\$PAS. CG\$CPC functions differently from CG\$PAS in that it fully processes each sequence and writes it to the SLIST file immediately, regardless of optimization level (no global optimizations are performed). The End Processor may intermix calls to CG\$CPC with calls to CG\$CUB to change local blocks as desired. It may also call CG\$DSA (Section 8.4.2) as necessary to reserve storage for variables.

5.1.2 Constants and Temporary Storage

The End Processor must call CG\$EP (Section 8.4.2) which outputs the used constants from CVT, reserves storage for the temporary arrays, OT. and IT., and modifies CUT for use by the Assembler. The host is responsible for the initialization code and storage reservation for the VD. array.

5.1.3 END Card

The End Processor must complete the SLIST file by writing an END card to it (see Chapter 6 for format).

[[] PAGE BREAK-- end-21-47

5.2 COMPLETING THE TABLES

The End Processor is responsible for completing LBT, Word C of the symbol tables and the AD table (if used). (AD OR A0 ??)

5.2.1 The Local Block Table

After all local code and variables have been written to the SLIST file, the End Processor must complete LBT by calling CG\$RBT (Section 8.4.2). This routine updates LBT from the entry points CC\$LBO, CC\$BLEN and CC\$PC, reformats LBT to contain program relocatable addresses as required by the Assembler, & computes the program length, leaving it in NS\$LBT.

5.2.2 Word C of the Symbol Table

The End Processor must supply all address information for every symbol table entry. In particular, it must change the address for all local symbols from block relative to program relocatable.

5.2.3 Temporary Storage

If stack storage is selected for temporary storage (Section 3.2.1) and references are made to stack frame lengths, then the AD table must be modified to contain final lengths. If COMPASS compatibility is being supported (Chapter 6), then for each A0 entry, an SMACRO call, "AD.n EQU length" should be output to the SLIST file.

The SMACROS deck should contain the definition

```
      EQU      SMACRO (S1,C2), LAB
      ENDS
```

5.3 ADJUSTING MEMORY LIMITS

After the End Processor is finished calling on the Code Transformer

it should release the code space that it occupies by calling CG\$IEP (Section 8.4.2).

[[] PAGE BREAK-- end-21-48

6.0 INTERNAL, ASSEMBLER

6.1 INTRODUCTION

The CCG Internal Assembler (CG\$IA) performs the functions that are usually performed by the second pass of a full assembler such as COMPASS. It reads the formatted instructions from the file SLIST, adds address information that is supplied through the symbol, block and constant tables, & produces a relocatable binary on the LGO file (if selected).

If the option is selected, it will also produce a formatted listing of the object code. An alternate option will produce a set of COMPASS source statements on the file COMPS. This file is suitable to use as input for a COMPASS assembly. The Assembler can produce either the object list or the COMPS file, but not both.

CG\$IA contains a limited macro capability that can be used for generating data words containing display coded, constant or relocatable address fields. It can also be used for producing some sequences of machine instructions in the object code if it is not embedded within the portion processed by the Code Transformer.

6.2 ASSEMBLER INPUT

Input to the Assembler comes from the following sources:

- SLIST is a file that contains machine instructions, pseudo instructions and macro calls. Each of these is usually contained in one word, although a second word might be used & some macro calls may require three words (see Section 6.3 for the format).
- TABLES generated by the host compiler and the CodeTransformer.
 - SYM - the main symbol table (see Section 2.1.1)
 - CBT - the common block table (see Section 2.2.1)
 - LBT - the local block table (see Section 2.2.1)
 - CUT - the constant usage table (see Section 2.2.2)
 - GLT - a special symbol table for generated symbols

There is provision for additional special symbol tables (see Section 2.1.1)

- MACROS A table of SMACRO definitions that is created by a COMPASS assembly of the SMACROS common deck at build time. This table is loaded as an integral part of CCG.
- Miscellaneous option definitions & controls provided by the host compiler (see Chapters 7 & 9).

[[] PAGE BREAK-- end-21-49

6.3 SLIST FILE

The first part of the file contains the initial group of pseudo instructions followed by the initial declaratives. This part is written to the SLIST file by the front end or the Bridge. The machine instructions are written to SLIST by the Code Transformer when it completes its optimizations. The End Processor then writes the final group of declaratives and passes control to the Internal Assembler which writes an end-of-record, rewinds the file and reads it as input.

The Internal Assembler leaves the file positioned at the end-of-record.

6.4 MACHINE INSTRUCTIONS

While the code transformer and the internal assembler are intended to be used as a single package, either one may be used without the other. Within the context of using the combined package this portion of the specification is internal instead of external specification.

It is possible to use the assembler without the code transformer or vice-versa. In these cases, this section is part of the external interface of the assembler or code transformer.

In either case, it does complete the specification of the SLIST file.

The following list of machine instructions is an alphabetical list of the instructions that will be accepted by the Assembler.

[[] PAGE BREAK-- end-21-50

SI	COMPASS	GROUP	NAME
AND	BXi XJ*Xk	A	And
CX	CXi Xk	0	Count bits
DFA	Dxi XJ*Xk	A	Double floating add
DFM	DXi XJ*Xk	A	Double floating Multiply
DFS	DXi XJ-Xk	A	Double floating subtract
DRL	RXi XJ or		
	SU3Q i,J»FP,R	P	Direct read LCM
DHL	WXi XJ or	\ or	
	SUBO. I,j,FP,W	P	Direct write LCM ./SUBO
EQV	BXi- Xk-XJ	C	Equivalence
FA	FXi XJ*Xk	A	Floating add
FO	FXi XJ/Xk	A	Floating divide
FM	FXi XJ»Xk	A	Floating multiply
FMA	MXi jk	E	Form mask
FS	FXi XJ-Xk	A	Floating subtract
IA	IXi XJ+Xk	A	Integer add
IAZ	IXi XJ+Xk	A	Integer add zero
ILD		H	Initial load
ILS	LXi Bj,Xk	B	Indexed left shift
IM	IXi XJ»Xk	A	Integer multiply
IMP	BXi -Xk-XJ	C	Implication
IRS	AXi Bj,Xk	B	Indexed right shift
IS	IXi XJ-Xk	A	Integer subtract
ISZ	IXi XJ-Xk	A	Integer subtract zero
JIN	JP Bi+k	J	Indexed Jump
JPBB		L	Conditional B Jumps
JPX		M	Conditional X Jumps
KLS	LXi Jk	E	Constant left shift
KRS	AXi Jk	E	Constant right shift
LD		H	Load
LDC		I	Load constant
LDV		I	Load Vardim
NR	NXi BJ,Xk	B	Normalize

OR	BXi	XJ+Xi	A	Or
PK	PXi	B],Xk	B	Pack
PLD			H	Parametric load
PST			H	Parametric store
PFA	RXi	XJ+Xk	A	Rounded floating add
RFO	RXi	XJ/Xk	A	Rounded floating divide
RFM	RXi	XJ»Xk	A	Rounded floating multiply
RFS	RXi	XJ-Xk	A	Rounded floating subtract
RJXJ			N	Conditional return lump
RJ3	RJ	k	K	Return Jump (30-bit)
RJ6			0	Return Jump (60-bit)
RNZ	ZXi	Bj,Xk	B	Rounded normalize
S			M	Set
SA			F	Short add
SOL			G	Short difference load
SOS			G	Short difference store
SLO			F	Short load
SS			G	Short subtract
SST			F	Short store
ST			H	Store

[[] PAGE BREAK-- end-21-51

STR	BXi	-Xk*Xj	C	Stroke (NAND)
STT				Store Transmit
TLD				Temp Load
TST				Temp Store
UJP	EQ	K	K	Unconditional Jump
UP	UXi	Bj,Xk	B	Unpack
XMT	BXi	Xj	D	Transmit
XHTC	BXi	-Xj	D	Transmit complement
XOR	BXi	XJ-Xk	A	EXCLUSIVE OR

The format of the instruction depends upon which group it belongs to, but there is a general format that is followed. It is the SI format and it is:

1/H2,11/OPC,18/CA,18/IH,/RJ,6/RI

H2 =1 if there is a second word for this instruction that contains a reference to a second entry in the main symbol table.

The second reference is in the low 18 bits of the next word and & It only occurs in instructions of group H.

OPC This field contains the IL Opcode that appears in the above list. It is biased by 2000B because it is inserted using a pack instruction.

CA Constant address or value.

IH Symbol table reference (Section 2.1.3 for reference format)

RJ The) register type and number

RI The i register type and number.

For the three instructions RJ3, UJP, and RJ6, the IH field is not contained in bits 12 through 29, but is in bits 0 through 17 instead.

Instruction groups A through G are, in most cases, instructions that use three registers. They are all single parcel instructions that do not have any symbol table references.

The low six bits of the IH field are used as an RK field for the type & number of the k register.

For instruction groups A through E the machine instruction opcode that is assembled = the IL Opcode.

Group A (AND, OR, XOR, FA, FS, OFA, DFS, RFA, RFS, IA, IS, FM, RFM, OFH, FO, RFO)

Also IAZ, ISZ and IN are processed as IA, IS, and OFM, respectively

These instructions each involve three X registers. The high order three bits of the RI, RJ and RK fields are ignored. The register numbers are taken from the low order three bits.

Group B. (ILS, IR, NR, RNZ, UP, PK)

The RI and RK fields represent numbers of X registers. RJ is a B register.

Again as the register types are known, the high order three bits of each of these fields is ignored.

[[] PAGE BREAK-- end-21-52

Group C (STR, IMP, EQV)

Like group A; RI, RJ, and RK represent 3 X registers. In this case though, the internal representation conforms to the common convention of representing the register operands in the order BXJ Xk(*,+,-)XJ. Consequently, the k register number is taken from the RJ field and the j register number is taken from the RK field.

Group D (XMT, XMTC, CX)

This group of instructions involve only two X registers. The register numbers for the J and k registers are both taken from RJ. The i register number is still taken from RI.

Group E (KLS, KRS, FMA)

This group of instructions involves only one X register and a six bit constant.

The constant is contained in the low order six bits of the CA field in the SI format.

It replaces the i and k register fields in the machine instruction format.

The i register number is taken from RI.

Groups F, G, H and I do not generate machine operation codes corresponding to their SI Opcode.

The SI Opcode determines to which group the instruction belongs.

All instructions within a group will generate the same set of machine operation codes.

The machine code that is generated for any group is dependent on the register type portion of the RI and RJ fields.

The register type is in the upper three bits of these six bit fields. 0 indicates a B register, 1 indicates an A register, and 2 indicates that an X register is used.

Group F (SLD, SST, SA)

Produces the set of instructions: SY1 ZJ+Bk

B, A, or X are substituted for both Y and Z.

Group G (SOL, SOS, SS)

Produces the set of Instructions: SYi Zl-Bk

B, A, or X is substituted for Y.

B or X is substituted for Z.

Group H (LO, ST, STT, PLO, PST, S, ILO, TLD, TST)

Produces the set of instructions: SYi Zl+k

B, A, or X are substituted for both Y and Z.

k * CA«>(IH)-(IH2)

CA is the contents of the CA field.

(IH) is the relative address taken from the symbol table entry IH.

If the H2 =1, a second symbol table pointer is taken from the following word

The relative address taken from that symbol table entry is (IH2).
When H2 is set IH and H2 both point to the main symbol table.
The resulting value is absolute.
If CA and IH are both zero, the group H instruction is changed to a group F instruction,
changing an SYi ZJ+0 to an SYi ZJ+B0.

[[] PAGE BREAK-- end-21-53

Group I (LDC, LDV)

Special load instructions for accessing the constant and Vardim tables respectively. Prior to processing the instructions as type H, the instruction processors change the CA of the instruction by the formula $CA = \text{low18}(\text{TBL}(\text{CA}))$ where TBL = CUT or VDI

See AAE directive in Section 6.6.2.4 for an alternate means of addressing a constant in the CVT.

Group J (JIN)

An unconditional Jump to (IH) as indexed by the B register specified in RI.

Group K (RJ3, UJP)

A return Jump & an unconditional Jump to (IH). For these instructions IH is taken from bits 0-17.

Group L (JPBB)

The CA field must contain a 0 - 3 to form the corresponding instructions EQ, NE, GE or LT.
This forms a conditional Jump (IH) using the B registers specified in RI and RJ.

Group M (JPX)

The CA field contains the three bit machine code modifier that determines the X register condition that is to be tested on the X register whose number is in low order three bits of the RI field.

Group N (RJXJ)

This conditional return Jump is functionally similar to the JPX except if the Jump is done it is a return Jump. This requires 2 instructions to be generated, a conditional Jump and a return Jump.
A force upper assures that the two instructions are placed in the same word.

The first is a conditional Jump to **1 that Jumps on the inverse of the condition specified by the modifier specified in the low order three bits of CA. The X register to be tested is specified in RI.
The return Jump is to (IH) plus an offset that can be specified in the upper twelve bits of CA.

[[] PAGE BREAK-- end-21-54

Group O (RJ6)

Return Jump with trace back. A force upper assures a new word, the upper half of which contains the return jump to (IH). The lower half contains twelve bits taken from CA and a reference to the external $\neq \text{HC\$RJTBN}$. IH is in bits 0-17. The expansion to a full word is performed by the Assembler.

Group P (DRL, OWL)

These direct LCM read and write instructions are implemented for the 7600 and CYBER 176 only.
They are fifteen bit instructions in the formats 0141J and 015iJ respectively.

The instructions are also used to indicate a level 0 address substitution if the IH field is non-zero.
In this case it holds the symtab ordinal of the formal Parameter.

RJXJ and RJ6 are forced upper before the instruction.
JIN, RJ3 and UJP are forced upper after the instruction.

6.5 SLIST PSEUDO IHSTRUCTIONS

The general SLIST pseudo instruction format is 12/P(OC),18/A,6/0,24/B.

References to the format are in the form "OC,A,B" where OC is the pseudo instruction name & A and B are the two operands in their order of occurrence.

The A and B fields will usually be designated as 0 to indicate that the field is not used, as SY to indicate that the field contains a symbol table ordinal in IH format

or as WC if it contains a word count. Even though 24 bits are allocated for the B field, its use is usually restricted to the 18 low order bits.

In the cases when 24 bits are allowed, the fact is noted in the description.

The pseudo instruction descriptions are divided into three groups that separate them according to where they can be located on the SLIST file.

6.5.1 Initial Group

This group of pseudo instructions must be grouped together on the SLIST file ahead of any other instructions for the subprogram. The first five may appear in any order and of the five only the IDENT is required. The USBLK is required as the last instruction of the group.

[[] PAGE BREAK-- end-21-55

LCC 0,WC Loader directive

LCC provides a means of including loader directives with the tables for a relocatable program.

The directive is contained in the following WC words on the SLIST file.

It is in display code, left-Justified with zero fill with at least twelve bits of trailing zeros.

IOENT 0,SY Program identification

SY is a pointer to the symbol table entry that contains the program name. The name is placed into the loader tables. The IOENT also causes the current time & date & control card options to be included in the binary prefix table.

COMNT 0,WC Prefix table comment

WC is the number of words following the COMNT word. These words contain comments that are to be included in the binary prefix table. There is a maximum of four words. The message is left-Justified and padded with blanks.

LIB 0L -> Libname

This instruction causes the specified library name to be added to those for which LIB= loader directives are to be written. Multiple LIB instructions are permitted in the initial group. The format of the LIB instruction is 12/P(OC.LIB),48/0L->Libname.

TITLE 0,WC Program listing title

The following WC words contain a heading title for the program listing.

The message is left-justified and padded with blanks.

USBLK 0,0 Terminate initial group

This instruction causes a set of initializations to be performed before processing the rest of the SLIST file.

LIB= directives for the library names specified by LIB pseudo instructions and the prefix table are output to the LGO file.

The binary control cards, local block and common block information

is put out on the initial page of the listing.
The symbol table is scanned to append a \$ to any symbol that would be confused with a register name. If the name is entered in the prefix table,
The \$ is not appended to the name in that table.

6.5.2 Declarative Group

These pseudo instructions are used in the declarative portions of the SLIST file.
This is following the initial group, ahead of the machine instructions and/or
after the machine instructions, ahead of the ENO pseudo instruction.

[[] PAGE BREAK-- end-21-56

USE 0,BN Define block to be used
BN is the CBT ordinal if the block to be used is a common block
or it is the LBT ordinal plus 2**15 if the block to be used is a local block.

BSS SY,WC Storage allocation
The BSS causes the current origin counter to be forced upper & a group of WC words
to be reserved. The content of the reserved words is not defined.
A SY \neq 0 is a symbol table pointer.
The address defined by the symbol table entry must be the same as the value of the
origin counter
after the force upper but before being incremented by WC. WC may be a 24-bit number.

BSSZ SY,WC Storage allocated and set to zero
BSSZ is the same as BSS with the additional provision that the allocated memory is
set to zero.

CON SY,wc Relocatable constant
SY is an ordinal into the symbol table. WC is added to the relocatable address from the
symbol table and stored in the low order end of a full word constant.
WC may be a 24-bit number.

OATA WC,CC Binary data
If WC = 0 the data must be less than 2**17 and is contained in the field CC.
If WC \neq 0 it specifies the number of data words that follow on the SLIST file.
Each of the words contains 60 bits of binary data.

DIS 0,WC Display coded data
Each of the WC words that follow contains 60 bits of display coded data.

HOL CC,F Display coded data left or right
CC is the number of display coded characters that is contained in the single word
that follows. The character string is left-Justified in that word on the SLIST file.
The format on the LGO file depends on the letter that is contained in F.
F = 18/1Rx where x = H,L,R
If x = H, the HOL is the same as the OIS with WC-1.
If x = L, the portion of the word to the right of the CC chars is cleared
If x = R, the characters are right-Justified and the left end is cleared.

[[] PAGE BREAK-- end-21-57

ORG SY,WC Set origin counter

There are three different forms of the ORG pseudo instruction.

They are:

Form 1. SY designates a symbol that has been previously defined.

Form 2. SY is zero.

Form 3. SY designates a symbol that has not been previously defined.

Forms 1 and 2 are used to manipulate the value of the origin counter.

There are two other special values that are saved by the Form 1 ORG.

They are K. and S.

K. is the maximum origin counter value.

It is saved by a Form 1 ORG for the use of a Form 2 ORG.

S. is the resulting origin counter value.

It is saved by a Form 1 ORG for the use of the REPI and VFOP pseudo instructions.

Form 1 causes the following steps to occur:

(1) If the current origin counter value is greater than K. it is saved in K.

(2) The origin counter is set to the address from the symbol table plus the 24-bit constant WC.

(3) The resulting origin counter is saved in S.

Form 2 causes the origin counter to be set from K.

Form 3 causes the symbol to be defined to the next location in the local block indicated by the micro HC\$UDVB (Section 9.3). It is equivalent to the following sequence:

```
USE *HC$UDVB
SY BSS 0
```

REPI DL,RC,INC,DA Instant replication

The REPI is a two-word pseudo instruction.

Word 1 = 12/P(OC),18/DL,12/0,18/RC. WORD 2 = 24/0,18/INC,18/DA.

The parameters are:

DL The number of words to be duplicated

PC The number of times that the data should be duplicated.

INC The increment between successive copies of the duplicated data.

DA The number of words to be added to the address established by the ORG.

The resulting address is where the first duplicated copy is placed.

REPI should always be preceded by an ORG.

ORG sets the origin counter to a relocatable address that is saved as a value called S.

DL words are copied from S. to S.+DA and repetitively to S.+DA+(n-1)*INC until n = RC.

The actual replication of data is performed by Loader.

CGIA produces the loader table that directs the process.

[PAGE BREAK-- end-21-58

DCS CC,I Define character string

The CGIA MACRO system provides for the definition of four macro strings that can be called from within a MACRO skeleton (Section 6.6.2.1). The DCS pseudo instruction provides a means for defining the string of characters that will be used when the macro is called.

DCS is a two-word pseudo instruction. The second word contains CC characters in 0L format.

These characters are saved as micro number I. CC is > 0 & < 10 while I = 1 to 4.

VFDP BN,LEN Partial word data load

The VFDP provides the capability of directing Loader to store a partial word field without disturbing the remainder of the word.

LEN bits of data is left-Justified in the second word of the pseudo instruction.

A loader table is written to LGO that will cause Loader to store the data in a word whose address (S.) is established by a preceding ORG.

The data is stored with its left-most bit at bit BN of the word. BN is a number 0 - 59.

6.5.3 END Instruction

This single instruction must be at the end of the subprogram.

ENO SYL,SYX End subprogram

SYL is the symbol table ordinal of the symbol for the LWA of the program

SYX is the symbol table ordinal of the symbol for the program transfer entry.

6.6 MACRO FACILITY

The limited macro facility which CG\$IA provides makes possible the construction of words containing various partial word fields. In order to use the macro facility, the host compiler must provide a set of macro definitions appropriate to its data structures. For clarity of discussion, these macro definitions are termed -SMACRO- definitions.

The SMACRO definitions are constructed using SMACRO directives which are provided by CG\$IA.

The SMACRO directives are simply macros written in COMPASS.

The directives, together with the SMACRO definitions, are assembled as an integral part of CG\$IA.

They result in tabular data which CG\$IA uses interpretively to expand SMACRO calls which appear on the SLIST file.

[[] PAGE BREAK-- end-21-59

6.6.1 SMACRO CALLS

An SMACRO call appears on the SLIST file in the format:

12/X0R(7777B,P(OC),24/A2,24/A1

12/0,24/A4,24/A3

12/024/A6,24/A5

where

OC opcode assigned is the SMACRO

P(OC) PACKED OPCODE

X0R(7777B,P(OC)) EXCLUSIVE OR of 7777B & P(OC) This is required to distinguish SMACRO opcodes from Machine & pseudo instruction opcodes.

Also SMACRO opcodes start at 128 so distinguished from machine that have the H2 bit set.

A1,A2,... argument values. The number of arguments & the type of each match the parameters specified in the SMACRO definition. Words used vary as required.

6.6.2 SMACRO definitions

An SMACRO definition consists of three parts: heading, body, and terminator.

Heading An SMACRO definition is headed by an SMACRO directive stating the name of the SMACRO and identifying the parameters in the SMACRO body.

Body The body of an SMACRO definition consists of SMACRO directives except SMACRO, ENDS, and DCSS. These directives make reference to the substitutable parameters. They may also make reference to micros, local symbols & global symbols.

Terminator An ENDS directive terminates an SMACRO definition.

6.6.2.1 Micros Local Symbols and Global Symbols

The directives within the body of an SMACRO definition refer to the substitutable parameters defined in the SMACRO directive. They may also refer to micro names, local and global symbols.

[[] PAGE BREAK-- end-21-60

MICROS

A limited micro facility is provided for passing string values whose lengths are greater than the 24 bit limit on an SMACRO argument. A micro table is defined within CG\$IA which contains entries for definition of four strings (M1,M2,M3, M4). Each string may be up to one word in length. The DCS pseudo-oo (Section 6.5.3) & the MIC directive (Section 6.6.2.3) are used to define the values of the strings (i.e., place the string value in the micro table). Within an SMACRO, the string value is obtained by referencing the appropriate entry in the table.

LOCAL SYMBOLS

An SMACRO definition may use up to 6 local symbols, designated as LI, L2,...,L6. A value is assigned to a local symbol by use of a SET directive (Section 6.6.2.3) or by appearing as the label on certain directives. A local symbol may assume a constant value or a relocatable value. CG\$IA contains a table with entries for LI to L6. When a constant value is assigned to Li, the value is retained in the corresponding entry in this table. When a relocatable value is assigned to Li, the corresponding entry contains a symbol table pointer which points to the CG\$IA abbreviated symbol table, SST. The relocatable value is retained in the appropriate entry in SST.

GLOBAL SYMBOLS

When it is desirable to reference a symbol from more than one SMACRO definition, a global symbol is used. A global symbol is defined by use of the DCSS directive. This directive causes reservation of a location (within CG\$IA) with the specified symbolic name.

NAME DCSS VALUE,TYPE

NAME is the symbolic name
VALUE is any legal COMPASS address expression. This parameter is optional.
TYPE sets the use of the symbol. SYM specifies that it contains a symbol table pointer in IH format. CON specifies that it contains a constant. CON is the default if this parameter is missing.

The value of the DCSS symbol may be set to an initial value by supplying the VALUE parameter. If the DCSS symbol is declared as an entry, its value may be set at run time by the host compiler (prior to execution of CG\$IA). The value & type of the DCSS symbol must be appropriate to its usage. If its type is SYM, the value must be a symbol table ordinal in IH format.

If its type is CON, the value must be a constant which is appropriate for the reference to it.

[[] PAGE BREAK-- end-21-61

Example:

```
ENTRY XYZ
XYZ   DCSS   3,SYM
```

XYZ becomes the name of a location whose initial value is 3.
Its type is declared to be symbol table ordinal.
It is declared as an entry, so the ENO Processor could change its initial value.
If not changed, all references to XYZ will become references to the third entry of the main symbol table.

6.6.2.2 Begin & End an SMACRO definition

The SMACRO & ENOS directives serve to mark the beginning & end of an SMACRO definition & to define the number and types of parameters within the definition.

```
NAME   SMACRO   PLIST,LAB
      ENDS
```

NAME is the macro name (≤ 6).
PLIST is a parenthesized list of parameter specifications (≤ 6).
LAB if present, must be the characters "LAB".
When present, it specifies that the first argument is to be placed in the label field when the macro is listed. In this case, the first parameter must be of type S.
PLIST is of the form (PI=ANAME,...).
=ANAME is optional. When present, ANAME may be any symbolic name & \neq ANAME may appear in the macro definition wherever the corresponding parameter may appear.

P specifies the argument type as follows:

P	Argument Type
C	24 bit constant
S	symbol table ordinal in IH format
B	block table ordinal which is either a CBT ordinal or LBT ordinal plus 2^{15} .
M	micro table ordinal. An M type parameter is used for listing purposes only. It is not passed into the body of the SMACRO.

I must be an integer 1 - 6. If the parameter is of type M then I must be 1 - 4 .
The I for each parameter must be chosen so that each parameter specification is unique.
For example, (C1,C2,M1,B1) and (C1,S2,M3,B4) are legal. (C1, C1) is not legal.

Any directive except DCSS, SMACRO & ENOS may appear between an SMACRO-ENDS pair.
Furthermore, any COMPASS pseudo-op which does not result in storage reservation may appear between the pair (e.g., conditional assembly instructions).

[[] PAGE BREAK-- end-21-62

The listing line produced when an SMACRO call is made is

```
ARG   NAME ARG,ARG,...
```

where ARG is dependent on parameter type as follows:

C actual constant
S symbol name from the symbol table
B block name from CBT (enclosed with slash marks) or from LBT
M character string from the micro table enclosed in parentheses.

6.6.2.3 Value Specifications

Two directives are provided for associating values with symbols.

The MIC directive associates the symbolic name in a symbol table entry with one of the micro names M1 to M4.

The SET. directive assigns a value to a local symbol or to a symbol table entry.

MI MIC FIRST-CHAR, N-CHAR, SEP-CHAR, SYMORD

MI is the micro identifier (M1 - M4). Note: its a direct reference to the micro table.
It is not interpreted as a substitutable parameter •

FIRST-CHAR is a constant which specifies which character in the symbol name should become the first character of the micro string. If the character specified by SEP-CHAR is encountered before FIRST-CHAR, the micro string becomes a null string.

N-CHAR is a constant which specifies the maximum number of characters that should be taken from the name to make up the micro string. If the character specified by SEP-CHAR is encountered before N-CHAR, the string will terminate with the char preceding the separator character.
The remaining characters through N-CHAR are replaced with blanks.
If N-CHAR=0, the string is not terminated by a length specification,
only by the separator character. N-CHAR <= 8, requirement.

SEP-CHAR is the separator character.

SYMORD specifies the symbol table entry from which the name should be obtained.
It must be an S type parameter or a DCSS symbol of type SYM.

Example!

M1 MIC 3,0,\$,S1

The name from the symbol table entry designated by the S1 parameter is used to construct a string which is to be associated with the micro M1.

Assume the name in the symbol table is "12ABC\$" .

The resulting micro string would be ABC.

[[] PAGE BREAK-- end-21-63

X SET. VALUE

X specifies the symbol with which the value is associated.
It may be a local symbol, an S type parameter (restricted to the main symbol table),
or a DCSS symbol of type SYM (restricted to the main symbol table).
The value is stored in Word C of the symbol table entry indicated by X unless X is a

local symbol & the value is absolute. In this case, the value is considered a constant & is stored in the CG\$IA table for local symbols (Section 6.6.2.1).

VALUE is an expression which, when evaluated, results in the value to be associated with X. An expression consists of a one or more elements joined by the operators +-*/. The operators joining elements are evaluated from left to right. An element can be a local symbol, an S parameter, a DCSS symbol, the origin counter (*), a constant, or a C parameter. The element value used is either the constant specified by the element or in the case of S type elements, is the content of Word C of the indicated symbol table entry. It is not legal for the expression, nor any sub-expression to result in negative relocation. Relocatable values may not be used with the * and / operators.

Examples:

L1 SET. C1-C2
L1 assumes a constant value which is the difference between the arguments C1 and C2

S1 SET. S1-2
Word C of the symbol table entry indicated by S1 is reduced by 2.

6.6.2.4 Define Part-Ward Fields

The BTW & ETW directives serve to delineate the beginning and end of the specification for a text word that contains part word fields. BTW specifies a template word which is used as the base of the definition. The ACI, ARI, & ASV directives serve to add constant, relocatable & string data to the template. They must appear between a BTW-ETW pair.

BTW ADDR

A00R is the address of the template word into which the part word fields are to be added. If A00R is absent, a zero word is used.

The BTW directive specifies the beginning of a text word definition. It always begins a full word definition and, hence forces upper. The only directives that may appear between a BTW-ETW pair are ACI, ARI, ASV, MIC, SET., IF.XX, ELSE., and ENOIF.

ETW

The ETW directive delineates the end of a text word definition.

[[] PAGE BREAK-- end-21-64

ACI TOP-BIT,BIT-LEN,CON or ACI RELOC,CON

TOP-BIT a constant specifying the left-most bit of the field.
BIT-LEN a constant specifying the length of the field.
CON specifies the constant value. It may be a C type parameter, a DCSS symbol of type CON or a local symbol with an absolute value.
RELOC is a shorthand notation for specifying TOP-BIT & BIT-LEN
U for upper relocation (bit 47, length 18)
M for middle relocation (bit 33, length 18)
L for lower relocation (bit 17, length 18)

The ACI directive causes the value of the constant specified by CON

to be algebraically added to the value existing in the specified field in the template word.

ARI TOP-BIT,BIT-LEN,SYM or ARI RELOC,SYM

TOP-BIT,BIT-LEN & RELOC are as specified for the ACI directive.

SYM specifies the relocatable value. It may be an S parameter, a DCSS symbol of type SYM, or a local symbol that has been assigned a relocatable value.
The ARI directive causes the relocatable value from Word C of the indicated symbol table entry to be algebraically added to the value existing in the specified field in the template word.

AAE TOP-BIT,BIT-LEN,SYMORO,CA or AAE RELOC,SYMORD,CA

TOP-BIT,BIT-LEN, & RELOC are as specified for the ACI directive.

SYMORO is the symbol table ordinal.
CA is the offset into the table.

The AAE directive is provided to permit the address of a constant from the constant table CVT to be added to the specified field in the template word.
If SYMORD is the CVT ordinal, CA is replaced by the corresponding CUT value.

[[] PAGE BREAK-- end-21-65

ASV TOP-BII,NC,MIC

TOP-BIT a constant specifying the left-most bit of the field.

NC a constant specifying the number of characters in the field.
If NC=0, the field length is assumed to be the same as the actual length of the micro string.
If NC is < length of the micro string, the string is truncated to NC characters.
If NC is > length of the micro string, the string is padded with blanks to NC characters.

MIC is a micro name (M1,M2,M3,M4).
Note that this is a direct reference to the micro table.
It is not interpreted as a substitutable parameter.

The ASV directive takes the specified characters from the micro string and ORs them into the specified field in the template word.

Examples

ABC SMACRO (S1,M1)
BTW
M1 MIC 1,7,,S1
ASV 59,7,M1
ARI L,S1
ETW
ENDS

DEF SMACRO (S1,C1)

```

BTW    TMPL
ARI    U,S1
ACI    L,C1
ETW
ENOS

```

```

TMPL   SBO    B2+0
       SBO    B2+0

```

The ABC SMACRO causes generation of a text word containing the symbolic name & relocatable address of a parametric symbol table entry. Note that the M1 in the header line is not used as a parameter. It is used only to obtain a value for the listing. The second argument in the call must contain a one.

The DEF SMACRO causes generation of a word containing
 SBO B2+relocatable address
 SBO B2«constant

```
[[ PAGE BREAK-- end-21-66
```

6.6.2,6 Full Word Definitions -----

The CON., BSS., and BSSZ. directives are provided for definition of full text words.

```
LAB    CON,SYM,WC
```

LAB may be a local symbol, an S type parameter or a DCSS symbol of type SYM.
 It assumes the current value of the org counter & may be referenced later in a SMACRO def.

SYM may be an S type parameter, a local symbol with a reloc value or a DCSS symbol of type SYM.

WC may be a constant, a C type parameter, a DCSS symbol of type CON, or a local symbol with a constant value.

The CON. directive produces a full word of text containing the relocatable address from Word C of the symbol table entry indicated by SYM plus the constant specified by WC.

```
LAB    BSS. WC
```

LAB and WC are as specified above for the CON. directive.
 The BSS. directive reserves the number of words of storage indicated by WC.

```
LAB    BSSZ. WC
```

BSSZ is the same as BSS. except that the reserved words are preset to zero.

6.6.2.7 Conditional Control -----

IF.XX, ELSE. & ENOIF. are provided for conditional control of the assembly process,

```
LAB    IF.XX P1,P2
LAB    ELSE.
LAB    ENOIF.

```

LAB is a label to distinguish IF.XX, ELSE & ENOIF. sequences.
xx is one of the conditions LT,LE,GT,GE,EQ,NE.
P1,P2 may be an S parameter, a C parameter, a DCSS symbol, a local symbol or a constant.

The IF.XX directive causes comparison of the two parameters for the condition and if true, the directives following the IF.xx are interpreted. NOTE that if the parameters are of S type the comparison is made on the symbol table pointers, not on the values in the symbol table.

[[] PAGE BREAK-- end-21-67

6.6.2.8 BLOCK Control

The USE. directive is provided for changing blocks.

USE. BN

BN may be a B parameter or one of the fixed local block names listed in HC\$FLB (Section 9.3)

6.6.3 SMACRO Opcodes

CG\$IA expects the SMACRO opcodes to start at a predefined base and to be assigned sequentially to the SMACRO definitions in their order of appearance in the deck SMACROS. The comdeck, SMOCDDEF (CCG PL), is provided to aid in the assignment of the opcodes. It sets the base for opcode definition, defines the opcode for each SMACRO name by defining the symbol M\$name and provides null definitions for the SMACRO directives. For defining opcodes, any straight COMPASS code in SMACROS (i.e., anything besides SMACRO directives) must be conditionally assembled out.

6.7 COMPASS COMPATIBILITY

If the host compiler supports the option of producing output which can be assembled by COMPASS, it must supply an assembly time macro text. This text must include:

- * Definitions of macros which are local to the host (i.e., transliterations of the SMACROS to COMPASS).
- Definitions of macros which are peculiar TO CCG.
These definitions are provided in the deck CGHCRMD (contained on the CCG PL)
- Definitions in HCDEFS (Section 9.3.1).

[[] PAGE BREAK-- end-21-68

7.0 CRADLE REQUIREMENTS

This section details the entry points (flags, routines, equs, etc.) that must be present in the host compilers cradle.

7.1 FETS & FILE BUFFERS (FET = FILE ENV TABLE, file buffer ptrs, first, in, out & Limit)

The host must supply FET's for the following:

File	FET name	USE	Mode needed
OUTPUT	F.OUT	List file	Test Mode
SLIST	F.SLST	CG pre-binary	All Modes
LGO	F.LGO	relocatable binary	HC\$IA \neq 0
COMPS	F.CMPS	BCD line images	HC\$IA \neq 0

The host is responsible for opening the above files at the beginning of compilation and closing them at the end.

The host must supply CIO file buffers for all of the above. (CIO = Circular I/O)

File buffers may not reside in dynamic storage.

LGO, COMPS & host compiler token stream file may share a common CIO buffer.

The recommended buffer size is 2003B words.

It is possible for LGO and COMPS to share an FET.

7.2 CONTROL STATEMENT OPTIONS AND FLAGS

Control statement option flags are of the form HO\$XX where XX is the control card option name. In general flags have the value of binary zero if the option is not selected (XX \ll 0 on the control statement).

Note that a minus zero will be treated as a zero in CCG.

HO\$B \neq	0	If LGO file to be produced
HO\$C \neq	0	If COMPASS will produce LGO
HO\$ER \neq	0	If error traceback information is to be output.
		If OPT \neq 2 CCG forces upper at the beginning of each code sequence.
		It outputs an SB0 B2 * line number at the beginning of each code sequence that begins in parcel zero. This code may be used in conjunction with other code that the host must generate, and an object time reprieve package to give the user the routine and line number that was being executed when an interrupt occurred.
HO\$OPT =	0-3	Compiler selected optimization level shifted 58 bits left (0, 1S58, 2S58).
HO\$UO \neq	0	If Unsafe Optimization selected.
		Permits prefetch optimization to be done even though the step is variable (Section 3.1.3).

[[] PAGE BREAK-- end-21-69

HO\$LCM =	LCM access mode 0 if direct (18 bit addresses) +1 if indirect (21 bit addresses) -1 if giant (21 bit addresses & 21 bit subscripts)
HO\$LO\$0	\neq 0 if object program (assembly) list selected
HO\$OATE	A two word array containing the date & time in display code. Set by calling the OATE and CLOCK system macros.
HO\$TIME	The second word of the array HO\$OATE.
HO\$PN	A cell that contains the current stack frame ordinal (see 3.2.1)•
HO\$RDP	A cell that contains the address of the reprieve dump processor routine.

HO\$CCOP A three word array (30H format) containing the display coded values of control card options that a user might wish to see listed in the load map.
The contents of HO\$CCOP is placed in the 77 table of the relocatable binary.

HO\$CSN A cell used to hold the line number or statement number of the statement that CCG is processing. Set by CCG, interrogated by the compile time reprieve package.
If a sequence contains more than one executable statement, CCG sets HO\$CSN to the first in the sequence.

HO\$PRGN Name of procedure that is being compiled.
less than 8 characters in 0L format, part of the two word array HO\$MSG.

```
ENTRY HO$MSG,HO$PRGN
HO$MSG OATA 10HCOMPILING
HO$PRGN OATA 0
```

N\$FERR Number of fatal errors in the current procedure.
May be incremented by CCG in case of memory overflow or assembly errors

N\$EXST Number of executable statements in the current procedure.
Supplied by the host, printed when OPT=2 in debug mode.

CC\$FT First Time whose initial value is 0. CCG sets it $\neq 0$ the first time it is called.

CP.AFLS Actual SCM field length
CP.NFLS Nominal SCM field length. Usually * CP.AFLS-8,
this may be made smaller prior to calling COMPASS to protect information in high core

[[] PAGE BREAK-- end-21-70

CR.AFLL Actual LCM field length

LCM.FL Alternate name for CP.AFLL
HO\$OFLL Origin of allocatable LCM. LCM below this value will not be touched by CCG.

HO\$MML LCM memory mode = 1S59 if reduce mode. May be set by the following
HOSMML = 0
MEMORY LCM,HO\$MML,R
HO\$MML = SHIFT(H0\$MML,58)

HO\$UFLL Amount of LCM memory used by the Job set.

HO\$MFLS Maximum SCM field length available to the compiler, shifted left 30.
To set this cell HOSMFLS = 30/Z/0
MEMORY SCM,HO\$MFLS,R

HO\$PMLS The largest field length used by CCG for the procedure being compiled.

FS\$TITL Subtitle buffer, 12 words long

LS\$TITL Length of BCD line in the subtitle buffer, initially = 1.

N\$FPS Number of formal parameters (*FTN* only). The use of this is yet to be specified.

CP.ERCT Bit 59 of this cell is the binary regardless flag. If it is set,
a relocatable object deck is to be put out when there are source language errors.
Not necessary if CGIA is not being used.

HOSLVL2 Non-zero if program contains LCM direct read and write references.
Flag is necessary only if host supplied memory reference expansion routine
references it.

The following 2 cells are necessary only if HC\$FPAS is nonzero.

CC\$SUB Non-zero if CT generated any "SUB" macro references.
CC\$SUB0 Non-zero if CT generated any level 0 "SUB" references.

The following cells are necessary if the host is using the VD facility
(VD. symbols defined in SYMØEFS).

N\$VD Number of VD. cells assigned, initially zero
S\$VD Symtab ordinal of VD

[[] PAGE BREAK-- end-21-71

The following cells are referenced when OPT=2 is used

HO\$ORD Cell containing FWA of a buffer in static storage that is used by OPT=2
as a paging buffer. Recommended size on the lower Cybers is 2000b words.
HO\$OBL The length of the OPT=2 paging buffer.
Ft\$BT FWA of LBT
Z\$LBT Length of LBT (number of fixed local blocks)
N\$LINES number of lines written on the current page.
CP.PS number of lines per page.

7.3 UTILITY ROUTINES

The below listed comdecks reside on the COMPASS common comdeck OLDPL.
Macro names referenced are defined in CPUTEXT.

COMDECK Entry Points	Function or Called by
COMCSYS	SYS= called by SYSTEM macro RCL = called by RECALL macro WNB= called by RECALL macro MSG = called by MESSAGE macro
COMCCDD	CDD= convert binary digits to decimal
COMCCOD	COD* convert binary digits to octal
COMCMVE	MVE= called by MOVE macro
COMCDXB	DXB= convert BCD digits to binary
COMCCIO	CIO= called by I/O function macros
COMCRDC	RDC= READC macro
COMCRDW	RDW= REAOW macro
COMCWTC	WTC= WRITEC macro
COMCWTW	WTW= WRITEW macro
COMCSFN	SFN= RJ sXSFN= (SPACE FILL NAME)

7.3.1 Compiler Utility Routines

There is another set of common decks that are intended for use by several different compilers. These will be located on the CCG PL. The content of these common decks will not be modified in any way unless the change has been approved by an appointed representative of each of the affected compiler development groups.

COMDECK Entry Points	Function
CCOMRPV	RPV= Compiler common reprieve package
COMAOEF	none DESCRIBE, DEFINE macro definitions
CCOMGCM	none common compiler macro definitions

[[] PAGE BREAK-- end-21-72

7.3.2 Host Supplied Utilities

The routines' FA\$LOL and FA\$NPG are provided by the host compiler.

FA\$LOL will list one line of output. The calling sequence is:
B6 = first word address of the output line
B7 = length of the output line
X7 = number of blank lines to be output before the line.

FA\$NPG - Start a new page on the listing

Appropriate page headings and subheadings are printed on the new page.

The following miscellaneous entry point must reside in the cradle:

HE\$ABT SX1 address of a BCD message & RJ HE\$ABT
to abort a compilation because of an unrecoverable system or hardware error

7.3.3 SCOPE 2 Comdecks

The following comdecks are the SCOPE 2 equivalents of the MACEIO listed above (COMCCI0 to COMCWTW).

FA=DEFS Macro definitions to be included in CMPLTXT and CCGTEXT
FA=SET set up FET, FTTs
FA=CLO Close a file.
FA=EOR Write a end-of-record
FA=FLSH Flush a file holding buffer.
FA=OPE Open a file.
FA=RDC READC macro
FA=RDW REAOW macro in FA=0EFS
FASRWX REWINO macro in FA*DEFS
FA=WTC WRITEC macro in FA=0EFS
FA^WTW WRITEW macro in FA*DEFS

7.4 DEBUGGING FACILIITES

The following debugging facilities must reside in the host's cradle & be available to CCG when a test mode compiler is built. Alt of the following may be obtained from the CCG OLDPL (see 9.2.1.4).

OUTPTK A deck to do FORTRAN formatted output.

[[] PAGE BREAK-- end-21-73

OUTPTK is assembled with CCG and the relocatable should be positioned by the host so that it is loaded with the cradle. SNAP routines are obtained from the COMPASS OLDPL. The IDP reference manual should be consulted for comdeck names and installation procedure. The deck also contains the FORTRAN callable utility routine REMARK, to issue a Dayfile message, and the routine G0T0ER. to satisfy references to it. The PRINT macro may be used to output formatted lines to the list file via OUTPTK.

SNAP Contains routines to save & restore all the registers, print the contents of the registers or memory, & FTN developed interactive debug facility. In addition to entry points mentioned in the previous sections SNAP requires the following:

HO\$SNAP will have bit 1S(59-1Ra) set, where "a- is any alphanumeric character (A-Z, 0-9) specified with the host SNAP control card parameter.

The alphabetics and numerics have separate meanings.
The alphabetics are generally used to selectively control debugging output.
For example, if the bit for the letter T is set, table allocation information may be printed.
The numerics are used to qualify the alphabetics.
CCG is assigned to the number two (2). When bit 30 (1S(59-1R2)) is set, CCG may use the remaining bits to determine if debugging output is to be produced.
If the bit for number 2 is not set, no debugging output will be produced from CCG.

FPA= routine to find the relative address of a routine.
FRA= is contained in the comdeck CCOMRPV which also contains a compile time reprieve package.

Macros to obtain register & core dumps are in the common deck DBG=MAC.
A description of the Interactive Oebug Facility is available from the FTN Project.

When a mode error occurs during compiler execution of the CCG overlay, the system should return control to the hosts reprieve package and if the CCG overlay was executing then the entry point CG\$PTC in CCG should be called by an RJ.

[[] PAGE BREAK-- end-21-74

7.4.1 Compile Time Reprieve Macros

RPVDEF - define fwa of routine for reprieve package.

This macro must be called at the beginning of each routine before any code has been generated.

RNAM RPVDEF ENAM

RNAM = routine name.

ENAM = entry point will be B-ENAM

If ENAM is null, then the macro defines the entry point by first stripping off any leading B= from RNAM and then prefixing a B= to the result max length of 5 chars.

RPVFWA - Define Entry for routine name/address table

RPVFWA NAM,FWA

ENTRY NAM = routine name

FWA - routine fwa

If null, B=xxxxx is used,

where xxxxx are the first 5 characters of NAM.

The following entry points must be defined.

(HO\$CSN) = current statement number in binary

(HO\$FVT) = FWA of a file vector table.

Format of the table is 42/0L->LFN,18/FET_AOORESS,

0 words are ignored, terminated by a word containing -1

(HO\$MSG) first word of =C format message "COMPILING name"

RPV= RPV system communication word in CCOMRPV.

This is the first word of the RPV exchange lump package.

FPA= a routine to find a routine relative address using the RPV Routine Name Address table described below. It is available for use at any time.

It is called with a jump (not an RJ).

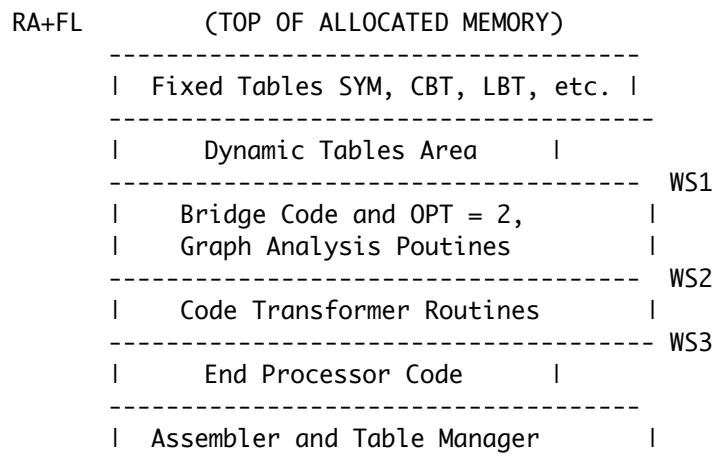
Its entry/ent conditions are described in the comdeck CCOMRPV.

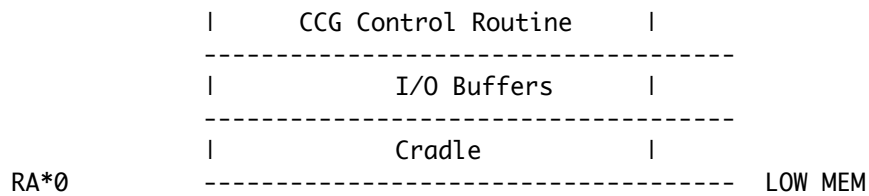
[[] PAGE BREAK-- end-21-75

8.0 OPERATIONAL ENVIRONMENT

8.1 MEMORY LAYOUT

If all pieces of the code generator are in one overlay,
then the memory layout is as depicted in the following diagram.





If it is desired to split CCG between two overlays, one may place the assembler (CGIA and MACROS) in a separate overlay. The table manager (CGTM) will have to reside in the cradle.

[[] PAGE BREAK-- end-21-76

8.2 WORKING STORAGE

The working storage area which is used for table management is increased as execution of the overlay progresses by releasing space occupied by code which is no longer needed. The initial working storage consists of core from F.MEM (WS1 in the above diagram) to FL. When OPT=2, the Global Optimizer releases the code space occupied by the Bridge and part of the Code Transformer (WS2 in the above diagram). The End Processor is responsible for releasing space occupied by the Code Transformer (WS3 in the diagram) after all code sequences have been processed CCG\$IEP is called to release the space).

8.2.1 F\$MEM

The host defines the static entry point F\$MEM as the first word of working storage after it initializes itself.

8.3 TABLE MANAGEMENT

The CCG table manager moves tables within working storage to obtain space for any given table. It will obtain more field length from the system as necessary to satisfy a space requirement. If more field length is not obtainable, the table manager increases N\$FERR by one and exits to either HE\$CTX, HE\$EPX or HE\$IAS as appropriate (Section 1.3).

The table manager considers tables to be either fixed or dynamic. Fixed tables are tables whose lengths do not change during execution of the overlay. They are packed together in high core and are moved as a group when FL changes. Dynamic tables are tables whose lengths may grow and shrink during execution of the overlay. The table manager moves them around in core on an as necessary basis. The reallocation algorithm used by the table manager is Garwick's algorithm as described in Knuth, Volume 1.

Two parallel vectors are used to maintain the locations and lengths of the tables. The first word addresses are maintained in the vector FTABE & lengths in the vector LTABE. The following symbols are defined for accessing individual entries in these vectors!

O\$XXX The location of the first word address of table XXX.

L\$XXX The location of the length of table XXX.

Z\$XXX The ordinal into FTABE and LTABE for table XXX.

The table manager depends on the fact that the order of the entries in the vectors reflects the actual ordering of the tables in memory. This order, from low to high memory, is defined to be:

[[] PAGE BREAK-- end-21-77

Table Group	Comdeck	supplied by
CCG Dynamic tables	CCGTMTV	CCG
Host Dynamic tables	CGHCSTD	Host
Host Static tables	CGHCSTD	Host

Each comdeck consists of a series of macro calls of the form:

NAME	TABLE	EQUIV
------	-------	-------

where Name [in column 2) is a 1 to 5 character table name.

A complete description of the TABLE macro may be found in the listing of CCGTMTV.

Certain tables are standard interface tables & must appear in one of the host supplied comdecks. They are -

CVT - Constant Value Table (Must appear in CGHCSTD)
CUT - Constant Use Table (Must appear in CGHCSTD)
CBT - Common Block Table
SYM - Symbol Table
GLT and other address definition tables

It is advantageous from a performance standpoint if as many host tables as possible are static.

The definition of the last dynamic table is provided by HC\$NQT which specifies the number of dynamic abbreviated symbol tables.

At load time the vectors are initialized so that the table first word addresses indicate the beginning of working storage and the lengths are zero.

The Bridge must initialize the table vectors & order the tables before calling CG\$INIT.
Specifically, it must:

- Pack all fixed tables in high core and record their addresses & lengths in FTABE & LTABE. This must include reserving space & recording the address & length (8) for the CCG table SST.
- Record the addresses & lengths of any dynamic tables which contain data (lengths are non-zero).
- Modify additional FTABE entries as necessary to assure that the address plus length for entry n is less than or equal to the address for entry n+1. This is required if dynamic tables containing data are interspersed with zero length dynamic tables. The addresses of the zero length tables must be initialized to reflect their placement between the actual addresses of the tables with data.

[[] PAGE BREAK-- end-21-78

The following operations on the tables are permissible:

- Allocate additional space for a table at the end of it via an ALLOC TNAM.LEN macro call which expands to

SAO =XZ\$TNAM	
SX1 LEN	LEN * contents (LSTNAM) /f0
RJ =XATS\$	

On exit from ATSS X0, X5, A5, B4, B5, & B7 are as before the call.

X1 = LEN
(A2,X2) = O\$TNAM, contents of O\$TNAM
(A3,X3) = L\$TNAM, contents of L\$TNAM, new length
(B6) = value of L\$TNAM prior to the call.

b. Add a single word to the end of a table by the macro call

ADDWRD TNAM,WORD which expands to
SAO =XZ\$TNAM
IFC NE./W0R0/X1/,1
SX1 WORD
RJ =XADW\$

On exit from ADW\$ one has the same exit conditions as AT\$ except
(X1) = (X6) = WORD

c. Reduce the length of a table by any of the following ways:

L\$TNAM = 0

or

L\$TNAM = L\$TNAM - CON

or

O\$TNAM = O\$TNAM+CON & L\$TNAM = L\$TNAM - CON

The static tables should be initialized by the algorithm FWA[I] a FWA[I+11 - LEN[I+11].

This is necessary since the table manager uses the FWA of the first static table as the LWA+1 of working storage for the dynamic tables.

d. Access/update an entry in a table. All accesses to a table must be made indirectly via the table vectors.

8.4 CCG ENTRY POINTS OF INTEREST

This section summarizes the entry points within CCG that are used by the host.

A section number appears below each entry point name which indicates the primary section in which the entry point is discussed. CCG will set B1 to 1 if it is required by the subroutine.

These subroutines do not use calling sequences that are compatible with SYMPL. (A CDC Language)
It is the responsibility of SYMPL coded Bridges or End Processors to provide interface routines to set registers where necessary.

8.4.1 Entry Points Referenced From the Bridge Only

These entry points are not available after the End Processor call CG\$IEP.

Name function and Calling Sequence

CG\$TNIT (X1=LBT ordinal of current block)

4.1 This routine is used to initialize the Code Transformer.

 XI must contain the LBT ordinal of the local block which is in effect on the SLIST file.

CG\$INIT initializes the table manager & places a skeleton BOS pseudo instruction in the table TXT, leaving L\$TXT = 4.

CG\$PAS This is the main entry point of the Code Transformer.

4.1 It is called each time the Bridge accumulates a code sequence.

CG\$LABD (X1^symtab ordinal)

4.4 Host calls CG\$LABD to define the block number associated with a label definition in OPT-2.

CG\$CPL (X6=LOC(TABL),X5=INOEX,B2=LEN)

4.5 returns (X1=INOEX to list in IOL, X2=0\$IOL, X0=LEN of list in IOL & X7=L\$IOL).

CG\$CPL is called to enter a list of names in the use/def dictionary and add them to IOL.

Initially the list is in a table that is pointed to by the cell TBL.

INDEX is the index into the table and LEN is the number of entries.

On the exit the list of use/def table indices have been added to the table IOL. OPT=2 only.

CG\$SCT (X1=CON),returns(X6=CRD)

2.2.2 Searches CVT for a previous occurrence of CON, adds it to the table if necessary & returns ORD to be used as the CA of a memory reference instruction.

CG\$ENC (X0=LOCF[TBL1,X5=N),returns(X6=ORD)

2.2.2 TBL is a vector of constant values, N words long.

CG\$ENC searches CVT for a previous occurrence of the constants in TBL, adds them to the table if necessary, & returns ORD to be used as the CA of the first constant in the vector.

CG\$FCU (X1=ORD)

2.2.2 Marks the constant in CVT at CVT(ORD) as used. It will be part of the final constant table.

CC\$PICL A flag, it may be used to hold the previous length of IOL.

4.5 It is reset to zero on exit from CG\$PAS. OPT=2 only.

[[PAGE BREAK-- end-21-80

CC\$OPTL A flag, it is non-zero on entry to CG\$PAS if TXT contains the body of an optimizable loop.

3.1.3 The flag is cleared on exit from CG\$PAS.

CC\$BRN A 2 word array holding the base & the limit R-numbers used in the sequence on

4.3.1 entry to CG\$PAS. On exit from it CC\$BRN(1) = CC\$BRN(2).

CC\$BIR A 2 word array with the base & limit intermediate R-numbers

4.3.1 (CC\$BIR(1) = 100002B) on entry) On exit CC\$BIR(2) = CC\$BIR(1).

CC\$NTRN Next intermediate R-number, this is another name for CC\$BIR(2).

4.3.1 Note that only CC\$BRN is referenced if HC\$ROL = 0 & the range of R-numbers on entry to CC\$BRN is between 4 and CC\$BRN.

CC\$SRF This entry point is used to indicate whether or not code sequence contains any references

3.2.2 which require address expansion (Section 3.2.2). If = 0, no address expansion is done, regardless of the setting off the AET field in the symbol table (Section 2.1.1.2).

The Bridge is responsible for initializing its value.

CC\$CBN When OPT=2, this entry point contains the current code sequence block number.

4.4 It is updated by the Bridge & used by the Code Transformer.

8.4.2 Entry Points Referenced From Bridge or End Processor

Name Function

AT\$\$ (AO=Z\$TNAM, X1*LEN)

8.3 Allocate table space

ADW\$ (AO-Z\$TNAM, X1*WØRØ)

8.3 Add a word to the end of the managed table

CG\$AVØ (X1=CA of a VD.), returns!X6=final CA]

This routine marks the CA'th VD. as materialized and assigns a final CA if this has not already been done. The final CA is saved in the CA field of the VDI table. The entry point N\$VD = the number of VD.s assigned.

CG\$CUB (X1 = LBT ordinal)

2.2.1 Used to change the local block in effect. It issues a USE declarative to the SLIST file, updates LBT from the content of CC\$LBØ, CC\$BLEN and CC\$PC and re-initializes these entry points to reflect the new local block.

CC\$LBØ entry point contains the LBT ordinal of the current local block.

2.2.1

[[PAGE BREAK-- end-21-81

CC\$BLEN This entry point contains the current length of the current local block.

2.2.1

CC\$PC This entry point contains the current parcel counter of the current local block.

2.2.1

CG\$RBT This routine uses the information in LBT, CC\$LBØ, CC\$BLEN, & CC\$PC to update

2.2.1 and reformat LBT so that it is in the state required by the Assembler.

It also provides the program length in N\$SLBT.

CG\$DSA (X1=IH, X2=NWDS)

5.1.1 This routine outputs "IH BSS NWOS" for the symbol and records its block relative address in the symbol table. IH must be a local symbol in the main symbol table only.

F\$RDT An array containing the descriptor words of the IL instruction opcodes.

4.3.5 SAI =XF\$RDT+ØC.opname fetches the descriptor word.

N\$SLBT A cell holding the sum of the local block 2.2.1 lengths,

4.3.5 it must be set prior to calling CG\$IA.

CG\$EP END PROCESSOR, THIS routine performs the following functions!

5.1.2 Outputs to SLIST a "CON. BSS Ø" followed by a DATA statement for each value in CVT whose corresponding CUT entry is non-zero.

Modifies CUT to contain the new ordinals of the constants for use by the Assembler.

Outputs to SLIST "OT. BSS length" and "IT. BSS length" to reserve storage for CCG generated temporaries.

Fills Word C in the symbol table for CON. , IT. & OT. Symbols to reflect their block relative addresses. Updates the block length of the current local block.

CG\$IA The entry to the internal assembler

6.0 MISC ROUTINES

SHL= (B7=FWA, X1=LEN) A simple shell sort for 1 word entry tables in ascending order.

CG\$PTC To print the contents of the compiler's tables in case of a compiler error only.

CG\$IEP Initialize table manager for end processing.

5.3 Causes all space occupied by Code Transformer code to be released to working storage.

[[] PAGE BREAK-- end-21-82

9.0 COMPILER BUILD ENVIRONMENT

9.1 GENERAL BUILD_PROCEDURE FOR A HOST COMPILER

CCG will always be assembled in the environment of the host. The general procedure is:

- a. Mount/attach the necessary OLDPL's which are the host's, CCG's & possibly COMPASS'S for the assembly of the cradle.
- b. Build the relevant texts which are
 ≠LPN≠TEXT for the host compiler, CMPLTXT for the host & CCG
 & CCGTEXT for CCG.
- c. Update & assemble the host.
- d. Update & assemble CCG.
- e. Create a suitable master binary to COPYL the assembled binaries onto and COPYL them.
- f. Generate the overlays & install them in system, etc.

9.2 SYSTEXTS

The assembly of the host and CCG will require the following texts for both NOS & NOS/BE:

CPUTEXT - system macros and symbols

IPTTEXT - hardware dependent installation options (referenced by CMPLTXT)

CMPLTXT - a text for compilers that use CCG it contains various useful macros,
 symbol definitions of compiler independent installation dependent options &
 the definitions of symbols that the bridge & the front end need to interface with CCG.

CCGTEXT - macros and symbol definitions for CCG

≠LPN≠TEXT - macros and symbol definitions for the host

9.2.1 CMPLTXT

TCMPLTXT contains the macro & symbol definitions that are common to both CCG & the host.

[[] PAGE BREAK-- end-21-83

9.2.1.1 Macro Definitions

PPVFWA Define entry tor Reprieve FWA table
RPVDEF define FWA of deck for Reorieve utility
LISTL to list one line on the output file
NUPAGE force eject & title a new page calls FA=LOL & FA=NPG in CCOMLOV

DEFINE, DESCRIBE, DEQU, REDEF, & BFMW macros for field definition.

9.2.1.2 Symbol Definitions

OC. symbols to define the SLIST & IL instruction opcodes
R1. & IH. symbols. to define IL instructions format
JC. symbols IL instructions format Jump codes for JPX. JPBB
SI. symbols SLIST instruction format
CF. symbols control flow table format
AP. symbols APLIST for at for use/def processing
FI. symbols function information word for RJ instructions
SO. symbols register specification fields
.CSET≠ 0 if 64 character set selected

IP.MFL maximum field length as set by the installation
.OS compile time op system = 1/2/3=for NOS, SCOPE 2, NOS/BE
,IWT instruction inter-word time = 1 if FC0 33261 installed on a 760
CP=RM compile time I/O system. 0 - use CI0, 7 - use 7RM
CT.ECS≠ 0 if ECS/LCM is available at compile time
OT=RM execution time I/O system
6-use 6RM, 7 - use 7RM
= 50/54. type of loader tables for compiler overlays
length of 0.0 overlay 50/54 table
length of non 0.0 overlay 50/54 table
CT.CPU value of compile time CPU
●CPU value of object time CPU
.DAL≠ 0 if RXX/WXX instructions available

The symbols IP.MFL through ,0AL are from the host supplied comdeck OPTIONS.

[[] PAGE BREAK-- end-21-84

9.2.1.3 Micro Definitions (from OPTIONS)

MDL\$	2 char string of execution time CPU
OS.NAME	op system name
OS.VER	op system version
SCM\$	CM or SCM
L CM\$	ECS or LCM

9.2.1.4 UPDATE Options

If the symbol TESTCCG is defined during UPDATE, a test version is generated.

9.3 COMDECKS SUPPLIES BY THE HOST COMPILER

- a. SMACROS - described in Chapter 6.
- b. SYMDEFS - this consists of WA., WB. and WC. definitions as described in Section 2.1.1.
It also contains the VD. definitions if that facility is used.
- c. HCDEFS - this is a collection of symbol definitions that describe the host to CCG.
Section 9.4 contains a listing of HCDEFS.
- d. OPTIONS - host compiler installation options [equs & micros]
see sections 9.2.1.2 & 9.2.1.3.
- e. CGHCSTD - a collection of TABLE macro calls for the dynamic tables that the host needs
during CCG processing
see section 8.3
- f. CGHCSTD - a collection of TABLE macro calls for the static tables that the host needs
during CCG processing.

9.4 HCDEFS

HC\$ID - Host compiler identification, possible values are -2 - FTN , 3 • PL/1

Used by CCG to conditionally assemble code in the following areas

Memory reference resolution (3.1.1),
Address expansion and resolution (3.2.2), and
Temporary storage assignment (3.2.1).

HC\$ID EQU 2

HC\$FLB - string of fixed local block names in the order that they
appear in "LBT" (Section 2.2.1)

HC\$FLB. MICRO 1, /START.,VAROIM.,ENTRY..CODE.,OATA., DATA..,HOL./

[[PAGE BREAK-- end-21-85

MC\$FPAS • ≠0 if formal parameter address substitution is used (Section 3.2.2)

MC\$FPAS EQU 1

MCSLNT - ≠0 if host is using the line number & symbol table features for
interactive debug (*FID*)

HC\$LNT EQU 1

HC\$MCIS - max chars in a symbolic name (≤8) that is in the symbol table (Section 2.1.1)

HCfMCIS EQU 7

HC\$RJJ - ≠0 if *RJJ* opcode is used by the host compiler (Section 4.3.3)

HC\$RJJ EQU 1

HC\$RJ6 - ≠0 if *RJ6* opcode is used by host compiler (Section 4.3.3).
HC\$RJ6 EQU 1

HC\$RJTBN - name of cell that holds the traceback information for a return Jump
(i.e. 42/7LNAME, 18/ENTRY A00R). (Section 4.3.1)
HCJRJTBN MICRO 1,,/TEMPAO./

HC\$ROL - ≠0 if R-numbers range over two sequences (Section 4.3.1) .
HC\$ROL EQU 1

HC\$STP - ≠0 if BR\$AFT is to be called from the code transformer
so the host can adjust the CA's of the ST.'s.

HC\$20C - ≠0 if CCG is to be set up for use in a two overlay mode

HC\$CTV - ≠0 if CCG shares the table manager vector with the host

HC\$IA - ≠0 if the host is using the CCG internal assembler.

HC\$UDVB - micro name of local block that usage defined variable* are placed in.
Used in *ØRG» pseudo (Chapter 6).
HC\$UDVB MICRO 1,,/DATA./

HC\$FRTP - abbreviated symbol table names. GL must occur as the first name (Section 2.1.2)
HC\$FRTP MICRO 1,,/GL,AP,IØ/

The host must also supply the values of the following micro*s
LPNAME\$ compiler name for use in title line, etc. This micro is 7 characters long.
LPNAME\$ MICRO 1,,/FTN /

[[] PAGE BREAK-- end-21-86

LPN\$ - truncated version that is placed in the 77 table,etc. Should be less than 7 characters
LPN\$ MICRO 1,,/FTN/

VERS - processor version, 3 characters.
VERS MICRO 1,,/4.6/

MODLVLS - PSR level of the host, 5 characters long,
MODLVLS MICRO 1,,/L439 /

[[] PAGE BREAK-- end-21-87 of IMS