UCID-21058

The Hybrid Compiler Writing System

Thomas A. Kuczmarski

May 1987

Lawrence Livermore National Laboratory

UCID--21058

DE87 010621

Acknowledgments

I wish to thank both Chuck Rasbold and Kelly O'Hair for the valuable contributions they have made to the design and practical utility of the hybrid compiler writing system. The system was designed concurrently with the construction of the first hybrid compiler (Model). Chuck worked on the Model front end and his opinions greatly influenced many of the details of the interface. Both Kelly and Chuck worked on the C front end and gave additonal input during that effort. I would also like to thank Kelly for his careful review of this document and for inspiring me to make it far more complete than I had originally intended.

Table of contents

Abstract

A set of compiler writing tools has been developed at the Lawrence Livermore National Laboratory which greatly facilitates the implementation of production quality compilers for a variety of languages. The tools are based on the optimization, code generation, and code scheduling portions of a highly optimizing, mature, preexisting Fortran compiler. The system is called the *hybrid compiler writing system*..

Introduction

The Civic compiler, implemented at the Livermore Computing Center (LCC) of Lawrence Livermore National Laboratory is a highly optimizing compiler which runs on the Cray family of supercomputers and compiles the LRLTRAN language (a dialect of Fortran). Several years ago it was decided to create a modular compiler writing system which would facilitate the implementation of new production quality compilers for additional languages by using the existing optimization, code generation, and code scheduling phases of the Civic compiler. Subsequent work led to the development of the *hybrid compiler writing system* which is the subject of this document.

Definition of a Hybrid Compiler

A *hybrid compiler* is a compiler (for some language) whose front end is written in an arbitrary language (not necessarily LRLTRAN), and which uses the Civic backend to perform the optimization, code generation, and code scheduling phases of compilation.

High Level Structure of the Hybrid Compiler Writing System

The hybrid compiler writing system consists of a set of LRLTRAN routines which provide an interface to the Civic backend, and two sets of run-time library routines used for loading the hybrid compilers themselves, and for loading programs compiled with a hybrid compiler. The interface routines are of necessity written in LRLTRAN because they must share the LRLTRAN definition files (called cliches) with the Civic backend routines. The run-time libraries are also written in LRLTRAN but they could have been written in any language.

Abstract

Scheduling portions of a

Design Criteria for the Hybrid Compiler Writing System

. **First and foremost, conceptual clarity**
The Civic compiler has been in existence for a long time and has been worked on by many different people at two different computing centers over an extended period of time. It was initially designed and implemented as strictly an LRLTRAN compiler (no block structure) and with no notion of accomodating multiple front ends for a variety of languages. A lot of low level detail is necessary in order to successfully create the Civic internal symbol tables and the internal intermediate language (called **the dag language**). It was necessary to figure out how to physically interface with the existing code and to find the proper conceptual level at which a variety of modular front ends (especially those that had to deal with block structure) would operate. A good deal of effort was expended in trying to determine the optimal level of abstraction for the interface routines. Too low a level would make each front end unnecessarily complicated as it reinvented the wheel creating common dag patterns to represent functionality commonly present in most higher level languages. On the other hand, too high a level of abstraction would presume knowledge of all the primitives any front end could possibly need. In general the interface routines represent higher level constructs than those expressed in the dag language. Hybrid front ends mainly deal in the "language" conceptually defined by the interface routines.

. **Callable from multiple languages (e.g., C and Model)**
In general, the Civic internal routines can only be called from LRLTRAN. A major reason involves the LRLTRAN "generic" feature which specifies a non-standard subprogram calling sequence. Another reason involves the LRLTRAN-specific definition files (cliches). These issues necessitated the creation of a set of LRLTRAN interface routines coded such that they could reasonably be called from a variety of languages.

. **General enough to handle multiple front ends which compile a variety of languages**
The interface routines are designed such that they have been successfully used to construct two hybrid compilers for languages as dissimilar as C and Model (basically a Pascal derivative with abstract data types). C is rather low level and contains no block structure, whereas Model is rather high level and does contain block structure.

. **Data hiding**
The interface routines successfully shield the front end designers from the myriad details necessary to successfully create Civic dags, symbol table entries, and auxiliary data structures. Thus the front end implementors can concentrate on the problems of compiling the new language rather than getting bogged down in unnecessary details which should be transparent at that level.

. **Easy to use (integrated with other LCC software)**
The interface routines were designed with other LCC software in mind. For example, the function call and formal/actual parameter mechanism expressed in the interface was designed to work in conjuction with the natural order in which productions are recognized by an LR parser. LCC's LR parsing system was used in the development of the C hybrid front end and fit together very nicely with the interface routines.

. **Make it possible to quickly implement new production quality compilers**
This has been demonstrated by the implementation of the Model and C hybrid compilers.

The Physical Structure of the Hybrid Compiler Writing System

The sources for the **hybrid interface routines** are contained in five source files named **intr1, intr2, intr3, intr4,** and **intr5.** (See **Appendix G** for where to find these files). Within these source files, there is a carefully written prolog immediately preceding each interface routine. Each prolog gives a very detailed explanation of the following:

Calling sequence:
Input:
Output:
Operation:
Special notes and warnings (when appropriate).

There is also an LRLTRAN cliche file called **intrclic (Appendix K)** which contains declarations of parameters and variables which are shared by the various interface routines. Intrclic also contains very detailed documentation of all these shared objects as well as which routines use them.

There are two run-time libraries, **hyblib** and **komnlib.** Hyblib is used for loading any program compiled with a hybrid compiler. Both hyblib and komnlib are used to load the hybrid compilers themselves. See **Appendix F** for a description of library dependencies and load order.

The sources for hyblib are contained in the following files:
minit, mzline, new, and rterr (See **Appendix I** for descriptions)

The Sources for komnlib are contained in the file:
mzget (See **Appendix J** for descriptions)

The Hybrid Interface Routines

The interface routines roughly break down into the following five categories:

. General communication with the Civic backend.
. Declaration of data objects
. Manipulation of data objects (words, parts of words, word vectors, and bit vectors).
. Subprogram control (including actual and formal parameters).
. Miscellaneous operations.

Appendix A contains a list of all the interface routines ordered as they appear in the source listings. **Appendix B** contains a list of the interface routines sorted alphabetically. Both of these lists name the source file for each routine and also a one line description of each routine. **Appendix C** contains a list of the interface routines which shows the formal parameters and the source line number where each routine starts.

Subroutines:

The file, **intr1**, contains the interface routines which are meant to be called as subroutines. Within this file, routines which have something in common tend to be next to each other.

All the routines in this file fall into one of the following categories:

. Backend Communication and Control:
(These are listed in the order that a front end would most likely call them).

setvrsn	- set compiler version number
setxfnam	- set executable file name
exeline	- set execute line
glinit	- global initialization
procinit	- procedure initialization (subprogram initialization)
backend	- execute civic backend for subprogram or main program
intrterm	- interface termination

. Object Declaration and Attribute Setting/Retrieval:

getoff	- get offset into stack frame for this data object
highuse	- high usage variable
nvectemp	- no vector temporary will be generated for this ple
nvhazrd	- no vector hazard
vhazrd	- vector hazard

See file, intr2, for related operations.

. Data Loading:

dataladr	- data load an address (load time computable)
datalval	- data load a value (compile time computable)

. Subprogram Call and Creation of Actual Parameters:

actlparm	- actual parameter (create an actual parameter)
spcal	- subprogram call (initialization)

See file, **intr5**, for the rest of the subprogram call routines.

. Special Operations:

qvar4kil	- queue variable for kill

. Hybrid Interface Internal (NOT TO BE CALLED BY FRONT ENDS):

intrerr	- NOT USED BY FRONT END (internal error routine)
q8return	- NOT USED BY FRONT END (dummy to satisfy the loader)
xcmplx	- NOT USED BY FRONT END (dummy to satisfy the loader)

Functions:

The file, **intr2**, contains miscellaneous routines which are meant to be called as functions. Within this file, routines which have something in common tend to be next to each other. Among other things, this file contains the interface routines for dealing with parts of words.

All the routines in this file fall into one of the following categories:

. Line number and file name debugging info:

file2reg	- address of location with ASCII for file name ---> reg
linc2reg	- line number ---> register
rvalfile	- right value of the file name
rvalline	- right value of the line number (register)

. Object declaration and object attribute retrieval:

comnvar	- symbol table entry for common variable in named common
getstp	- get symbol table pointer for an object
inlfname	- symbol table entry for an inline function name
litval	- make a literal table entry for a numeric constant
localvar	- make a symbol table entry for a local variable
makelabl	- make symbol table entry for a used label
makext	- make a symbol table entry for an external object
namecom2	- declare a named common block
strcon	- make a literal table entry for a string constant

See intr1 for related operations.

. Partial word manipulation

bextract	- byte (partial word) extract
binsert	- byte (partial word) insert (type 1)
bitsb4	- bits before (bits preceding a byte in a word)
byteoff	- compute byte offset
dbinsert	- byte (partial word) insert (type 2)
wdoffset	- compute word offset

. Special operations for constructing the civic intermediate language.

asmcode	- build dags for inline assembly code
bbreak	- insert a block break
sdage	- store your own custom made dag
sple	- store a ple (program list element)

. Special Operations

cardinal	- number of 1 bits in a vector
fbit	- position of first 1 bit in a boolean array (type 1)
fbitaddr	- position of first 1 bit in a boolean array (type 2)
kilvars	- kill stores to variables

. Hybrid Interface Internal (NOT TO BE CALLED BY FRONT ENDS):

stopval	- for testing new dags

The file, **intr3**, contains routines which return either the right value, left value, or address of the following basic data objects:

. words	**(rval,**	**lval)**
. word vectors	**(rvalvec,**	**lvalvec)**
. bit vectors	**(rvalbv,**	**lvalbv)**
. any of the above	**(address,**	**addrext)**

The functions of these routines are encoded in their names as follows:

rval	- right value (word)
rvalvec	- right value vector (word vector)
rvalbv	- right value bit vector
lval	- left value (word)
lvalvec	- left value vector (word vector)
lvalbv	- left value bit vector
address	- address (of anything)
addrext	- address of an external

The file, **intr4**, contains routines which return either the right value, left value, or address of the following data objects:

1) Temporaries internal to the hybrid interface:

a) words	**(rvalt,**	**lvalt)**
b) word vectors	**(rvaltvec,**	**lvaltvec)**
c) bit vectors	**(rvaltbv,**	**lvaltbv)**
d) any of the above	**(addrt)**	

2) Objects that are the result of dereferencing a pointer expression:

a) words	**(rvali,**	**lvali)**
b) word vectors	**(rvalivec,**	**lvalivec)**
c) bit vectors	**(rvalibv,**	**lvalibv)**
d) any of the above	**(addri)**	

3) Vector temporaries created by the backend **(rvalvt, addrvt)**
This file also contains two routines **(eqvec, neqvec)** which return dags which represent the value of an equal or not equal comparison between two vector objects (either word vectors or bit vectors).

The functions of these routines are encoded in their names as follows:

rvalt	- right value temporary (word)
rvaltvec	- right value temporary vector (word vector)
rvaltbv	- right value temporary bit vector
lvalt	- left value temporary (word)
lvaltvec	- left value temporary vector (word vector)
lvaltbv	- left value temporary bit vector
addrt	- address temporary
rvali	- right value indirect
rvalivec	- right value indirect vector (word vector)
rvalibv	- right value indirect bit vector
lvali	- left value indirect
lvalivec	- left value indirect vector (word vector)
lvalibv	- left value indirect bit vector
addri	- address indirect
eqvec	- equal vector
neqvec	- not equal vector

The file, **intr5**, contains routines which are used for handling formal and actual parameters, calling user defined subprograms, calling LRLTRAN intrinsic functions, calling a few hybrid special functions (new, dispose, and mzline), and getting the value of the real time clock.

All the routines in this file fall into one of the following categories:

. Declare Formal Parameters

ffmlparm	- create an internal "fake" formal parameter
fmlparm	- create a formal parameter

. Make a Subprogram Call and Return

endintrn	- terminate generation of lrltran intrinsic function call
endspcal	- terminate generation of a subprogram call (type 1)
endspdag	- terminate generation of a subprogram call (type 2)
expparm	- obsolete
intrnsc	- lrltran intrinsic function check
rtn	- return from a subprogram

See spcal and actlparm in file intr1 for registering actual parameters.

. Hybrid Special Functions:

calldisp	- generate a call to the hyblib routine, dispose
callmzln	- genrate a nonstandard call to hyblib routine, mzline
callnew	- generate a call to the hyblib routine, new
rtclock	· get value of the real time clock

The Run-Time Library Routines

The run time library routines break down into two groups. The first group is kept in the public file, **hyblib**, and contains the routines used for loading all hybrid compiled programs. These routines are summarized in **Appendix D**. The second group is kept in a private file called **komnlib**. These routines are used for loading the hybrid compilers themselves. These routines are summarized in **Appendix E**.

The Structure of a Hybrid Front End

Basically, a hybrid front end builds a set of data structures which represent the program to be compiled, and then transfers control to the Civic backend to perform the optimization, code generation, and code scheduling phases of the compilation. These data structures consist of the Civic internal symbol table entries, a ple/dag structure (a representation of the specific program in the dag language), and auxiliary structures such as common block tables and data load tables. The ple/dag structure consists of a linear chain of ple's each of which roughly corresponds to a single statement. Each ple points to a directed acyclic graph (dag). These dags describe the details of the program being compiled. Generally, dags are built bottom up and ple's are inserted at the end of the chain. However, the ple chaining mechanism allows a ple to be inserted at an arbitrary point in the ple chain and this is sometimes necessary.

The front end calls the interface routines to perform the following functions:
. Miscellaneous optional initializations
. Initialize the Civic backend.
. Process subprograms sequentially.

Appendices L and M show the exact sequence of interface calls that were actually produced by the C and Model hybrid compilers working on sample programs. Here is a rough outline of what interface routines might be called by a front end.

sethprms - set front end heap expansion parameters (optional)
stopheap - stop front end restricted heap from expanding further
glinit - initialize the Civic backend

procinit - procedure initialization (once per subprogram)

. - other interface routine calls
.

backend - execute the Civic backend for this procedure (once per subprogram)
getoff - for block structured languages only. Get offset for a local variable within current
. stack frame (these offsets are assigned by the Civic backend)
.
.
getoff

intrterm - compiler termination

A hybrid front ends uses the system memory manager to manage the space for its heap. However, because the Civic backend manages its own memory, the system memory manager

must use a restricted heap when satisfying requests for the front end. This is all handled automatically when you load a hybrid compiler with komnlib (**Appendix E**). The prologs to the komnlib routines sethprms, stopheap, and mzget (**Appendix J**) give a complete explanation of how this is implemented and what rules a front end must follow in order to coexist with this mechanism. The rules are fairly simple.

Miscellaneous rules and things to watch out for when coding hybrid front ends.

. If you are building a front end for a block structured language, you must maintain a block structured symbol table (separate and distinct from the Civic internal symbol table) in the front end. The Civic symbol table is completely flat and has not the slightest notion of block structure. In addition, the Civic symbol table is completely reinitialized for each subprogram.

. Use IOC's greater than 15. The Civic backend uses IOC numbers 15 and less.

. It is thoroughly documented in each relevant interface routine prolog, but occasionally overlooked that the formal parameter called "name" is both an input and output parameter. Name, always contains a sequence of printable ASCII characters. Because of restrictions within the Civic backend, "name" must be blank filled out to a word boundary. Thus the interface routines will always space fill the formal parameter, "name", before using it (just for safety). If the calling routine is not aware of this, data in the caller may be overwritten with a few ASCII spaces. See interface routine rval for an example.

How to Load Hybrid Compilers

Loading hybrid compilers is a little involved because some tricks are used to decrease the compiler size and because the relocatable binaries which are used come from a variety of sources. Thus, self-documenting cosmos controllers are always used to load the C and Model hybrid compilers. These controllers change somewhat from time to time, so rather than reproducing them in this document, we will merely state where they can be found.

The cosmos controller to load the C hybrid compiler is called **genv** and is located in public file, newccomp.

The cosmos controller to load the Model hybrid compiler is called **mkltss** and is located in public file, mod.

By examining these files when you need to, you will always have the current version of exactly how things are being done.

Clones:
And now we get to the painful issue of clones. As discussed in **Appendix G**, the hybrid interface and Civic backend binaries live in a build library file called something like bin900<letter> or bin901<letter>. This file contains binaries for a hybrid compiler which will execute under the LTSS operating system and generate code for a Cray-1 (also works on a Cray XMP), using non-unicos binaries.

However, that's not good enough. So we have all kinds of special purpose versions of the bin900's which we call *clones*. These are used to load all kinds of special purpose versions of the hybrid compilers. Here's a description of what we've got so far, and what the naming conventions are. By the time you read this, no doubt, the situation will become more complicated.

Description of Clone	Sample Name	Cosmos Deck to Generate the Clone
ltss /cray1/oldbins	bin900t	crbin
nltss/cray1/oldbins	bin900tn (nltss)	crbinn
ltss /xmp48/oldbins	bin900tx (xmp)	crbinx
nltss/xmp48/oldbins	bn900tnx	crbinnx
ltss /cray1/unicos	bin900tu (unicos)	crbinu

The cosmos decks that start with the letters cr are kept in public file mod.

compiler size and because the relocatable binaries

Appendices:

APPENDIX A: INTERFACE ROUTINES in Naturally Occurring Order (with one line descriptions).

APPENDIX B: INTERFACE ROUTINES Sorted Alphabetically (with one line descriptions).

APPENDIX C: INTERFACE ROUTINES with Source File Line Numbers and Formal Parameter List.

APPENDIX D: HYBLIB - Run Time Routines for Loading User Programs.

APPENDIX E: KOMNLIB - Additional Run Time Routines Used to Load the Hybrid Compilers Themselves.

APPENDIX F: Library Dependency Diagram

APPENDIX G: Where the Files are Stored (public files and xport directories).

APPENDIX H: INTERFACE ROUTINE Prologs.

APPENDIX I: HYBLIB Routine Prologs.

APPENDIX J: KOMNLIB Routine Prologs.

APPENDIX K: INTRCLIC - Master Cliche for the Interface Routines.

APPENDIX L: Sample Model Language (A Pascal Derivative) Program and Trace of How the Interface Routines Were Called.

APPENDIX M: Sample C Language Program and Trace of How the Interface Routines Were Called.

APPENDIX A

Interface Routines in Naturally Occuring Order
(with one line descriptions)

setvrsn	intr1 - set compiler version number
setxfnam	intr1 - set executable file name
exeline	intr1 - set execute line
glinit	intr1 - global initialiazation
procinit	intr1 - procedure initialiation (subprog initialization)
namecomn	intr1 - obsolete
spcal	intr1 - subprogram call (initialization)
actlparm	intr1 - actual parameter (register an actual parameter)
qvar4kil	intr1 - queue variable for kill
qft4kil	intr1 - obsolete
datalval	intr1 - data load a value (compile time computable)
dataladr	intr1 - data load an address (load time computable)
nvectemp	intr1 - no vector temporary will be generated for this ple
vhazrd	intr1 - vector hazard
highuse	intr1 - high usage variable
nvhazrd	intr1 - no vector hazard
backend	intr1 - execute civic backend for subprogram or main prog
getoff	intr1 - get offset into stack frame for this data object.
q8return	intr1 - NOT USED BY FRONT END (dummy to satisfy the loader)
xcmplx	intr1 - NOT USED BY FRONT END (dummy to satisfy the loader)
intrerr	intr1 - NOT USED BY FRONT END (internal error routine).
intrterm	intr1 - interface termination
line2reg	intr2 - line number -> register
file2reg	intr2 - addr of location with ascii for file name -> reg
rvalline	intr2 - right value of the line number register
rvalfile	intr2 - right value of the file name
makelabl	intr2 - make symbol table entry for a used label
bbreak	intr2 - insert a block break
makext	intr2 - make a symbol table entry for an external object
localvar	intr2 - make a symbol table entry for a local variable
namecom2	intr2 - declare a named common block
comnvar	intr2 - symbol table entry for common var in named common
inlfname	intr2 - symbol table entry for an inline function name
sple	intr2 - store a ple (proragm list element)
litval	intr2 - make a literal table entry for a numberic constant
strcon	intr2 - make a literal table entry for a string constant
asmcode	intr2 - build dags for inline assembly code
sdage	intr2 - store your own custom made dag
bextract	intr2 - byte (partial word) extract
binsert	intr2 - byte (partial word) insert (type 1)
dbinsert	intr2 - byte (partial word) insert (type 2)
byteoff	intr2 - byte offset
wdoffset	intr2 - word offset

bitsb4	intr2 - bits before (bits preceding a byte in a word)
fbit	intr2 - position of first 1 bit in a boolean array (type 1)
fbitaddr	intr2 - position of first 1 bit in a boolean array (type 2)
cardinal	intr2 - number of 1 bits in a bit vector
kilvars	intr2 - kill stores to variables
kilfts	intr2 - obsolete
getstp	intr2 - get symbol table pointer for an object
stopval	intr2 - NOT USED BY FRONT END (for testing new dags).
rval	intr3 - right value (word)
rvalvec	intr3 - right value vector (word vector)
rvalbv	intr3 - right value bit vector
lval	intr3 - left value (word)
lvalvec	intr3 - left value vector (word vector)
lvalbv	intr3 - left value bit vector
address	intr3 - address (of anything)
addrext	intr3 - address of an external object
rvalt	intr4 - right value temporary (word)
rvaltvec	intr4 - right value temporary vector (word vector)
rvaltbv	intr4 - right value bit vector
rvalft	intr4 - obsolete
lvalt	intr4 - left value temporary (word)
lvaltvec	intr4 - left value temporary vector (word vector)
lvaltbv	intr4 - left value temporary bit vector
lvalft	intr4 - obsolete
addrt	intr4 - address of temporary
rvali	intr4 - right value indirect (word)
rvalivec	intr4 - right value indirect vector (word vecor)
rvalibv	intr4 - right value indirect bit vector
lvali	intr4 - left value indirect (word)
lvalivec	intr4 - left value indirect vector (word vector)
lvalibv	intr4 - left value indirect bit vector
addri	intr4 - address indirect
rvalvt	intr4 - right value vector temp (temp created by backend)
addrvt	intr4 - address vector temp (temp created by backend)
eqvec	intr4 - dag representing test for equality between vectors
neqvec	intr4 - dag represents test for inequality between vectors
fmlparm	intr5 - register formal parameter
ffmlparm	intr5 - register an internal "fake" formal parameter
endspcal	intr5 - terminate generation of a subprogram call (type 1)
endspdag	intr5 - terminate generation of a subprogram call (type 2)
intrnsc	intr5 - lrltran intrinsic function call (initialization)
endintrn	intr5 - terminate generation of lrltran intrinsic func call
expparm	intr5 - obsolete
rtn	intr5 - return from a subprogram
callnew	intr5 - generic call to the hyblib routine called new
calldisp	intr5 - generic call to the hyblib routine called dispose
callmzln	intr5 - nonstandard call to hyblib routine, mzline

bitsb4 intr2 - bits before (bits preceding

rtclock intr5 - get value of the real time clock

rtclock intr5 - get value of the real time clock

APPENDIX B

Interface Routines Sorted Alpahbetically
(with one line descriptions)

actlparm	intr1 - actual parameter (register an actual parameter)
address	intr3 - address (of anything)
addrext	intr3 - address of an external object
addri	intr4 - address indirect
addrt	intr4 - address of temporary
addrvt	intr4 - address vector temp (temp created by backend)
asmcode	intr2 - build dags for inline assembly code
backend	intr1 - execute civic backend for subprogram or main prog
bbreak	intr2 - insert a block break
bextract	intr2 - byte (partial word) extract
binsert	intr2 - byte (partial word) insert (type 1)
bitsb4	intr2 - bits before (bits preceding a byte in a word)
byteoff	intr2 - byte offset
calldisp	intr5 - generic call to the hyblib routine called dispose
callmzln	intr5 - nonstandard call to hyblib routine, mzline
callnew	intr5 - generic call to the hyblib routine called new
cardinal	intr2 - number of 1 bits in a bit vector
comnvar	intr2 - symbol table entry for common var in named common
dataladr	intr1 - data load an address (load time computable)
datalval	intr1 - data load a value (compile time computable)
dbinsert	intr2 - byte (partial word) insert (type 2)
endintrn	intr5 - terminate generation of lrltran intrinsic func call
endspcal	intr5 - terminate generation of a subprogram call (type 1)
endspdag	intr5 - terminate generation of a subprogram call (type 2)
eqvec	intr4 - dag representing test for equality between vectors
exeline	intr1 - set execute line
expparm	intr5 - obsolete
fbit	intr2 - position of first 1 bit in a boolean array (type 1)
fbitaddr	intr2 - position of first 1 bit in a boolean array (type 2)
ffmlparm	intr5 - register an internal "fake" formal parameter
file2reg	intr2 - addr of location with ascii for file name -> reg
fmlparm	intr5 - register formal parameter
getoff	intr1 - get offset into stack frame for this data object.
getstp	intr2 - get symbol table pointer for an object
glinit	intr1 - global initialiazation
highuse	intr1 - high usage variable
inlfname	intr2 - symbol table entry for an inline function name
intrerr	intr1 - NOT USED BY FRONT END (internal error routine).
intrnsc	intr5 - lrltran intrinsic function call (initialization)
intrterm	intr1 - interface termination
kilfts	intr2 - obsolete
kilvars	intr2 - kill stores to variables
line2reg	intr2 - line number -> register
litval	intr2 - make a literal table entry for a numberic constant

localvar	intr2 - make a symbol table entry for a local variable
lval	intr3 - left value (word)
lvalbv	intr3 - left value bit vector
lvalft	intr4 - obsolete
lvali	intr4 - left value indirect (word)
lvalibv	intr4 - left value indirect bit vector
lvalivec	intr4 - left value indirect vector (word vector)
lvalt	intr4 - left value temporary (word)
lvaltbv	intr4 - left value temporary bit vector
lvaltvec	intr4 - left value temporary vector (word vector)
lvalvec	intr3 - left value vector (word vector)
makelabl	intr2 - make symbol table entry for a used label
makext	intr2 - make a symbol table entry for an external object
namecom2	intr2 - declare a named common block
namecomn	intr1 - obsolete
neqvec	intr4 - dag represents test for inequality between vectors
nvectemp	intr1 - no vector temporary will be generated for this ple
nvhazrd	intr1 - no vector hazard
procinit	intr1 - procedure initialiation (subprog initialization)
q8return	intr1 - NOT USED BY FRONT END (dummy to satisfy the loader)
qft4kil	intr1 - obsolete
qvar4kil	intr1 - queue variable for kill
rtclock	intr5 - get value of the real time clock
rtn	intr5 - return from a subprogram
rval	intr3 - right value (word)
rvalbv	intr3 - right value bit vector
rvalfile	intr2 - right value of the file name
rvalft	intr4 - obsolete
rvali	intr4 - right value indirect (word)
rvalibv	intr4 - right value indirect bit vector
rvalivec	intr4 - right value indirect vector (word vecor)
rvalline	intr2 - right value of the line number register
rvalt	intr4 - right value temporary (word)
rvaltbv	intr4 - right value bit vector
rvaltvec	intr4 - right value temporary vector (word vector)
rvalvec	intr3 - right value vector (word vector)
rvalvt	intr4 - right value vector temp (temp created by backend)
sdage	intr2 - store your own custom made dag
setvrsn	intr1 - set compiler version number
setxfnam	intr1 - set executable file name
spcal	intr1 - subprogram call (initialization)
sple	intr2 - store a ple (proragm list element)
stopval	intr2 - NOT USED BY FRONT END (for testing new dags).
strcon	intr2 - make a literal table entry for a string constant
vhazrd	intr1 - vector hazard
wdoffset	intr2 - word offset
xcmplx	intr1 - NOT USED BY FRONT END (dummy to satisfy the loader)

APPENDIX C

Interface Routines with Source File Line Numbers and Formal Parameter List
(sor900w, 4-21-87)

INTR1

```
  60      entry setvrsn(name)
  98      entry setxfnam(name)
 140      entry exeline(text, numwords)
 200      entry glinit(addr, srcfile, binfile, listfile, listopts, object,
 201     *            optimizations, trace, arithmetic, statistics,
 202     *            vectorization, vtmanagement, sensitive_ext, language,
 203     *            ema_parm)
 732      entry procinit(staticlevel, startline, orignaine, onumchars,
 733     *              genname, gnumchars, prgtype, datatype, namflag, locflag,
 734     *              mzlineflag, isparent, display_needed, firstple, rterrple)
1625      entry namecomn(name, numchars, bitsize, common_type)
1710      entry spcal
1766      entry actlparm(dagptr)
1820      entry qvar4kil(name, numchars)
1940      entry datalval(symptr1, wdoff1, bytesize, bitsbefore, value)
2025      entry dataladr(symptr1, wdoff1, bytesize, bitsbefore, symptr2, wdoff2)
2135      entry nvectemp(pleindex)
2178      entry vhazrd(pleindex)
2222      entry highuse(name, numchars)
2281      entry nvhazrd(pleindex)
2329      entry backend(startline, numlines, temp1siz, temp2siz, highfast_temp,
2330     *            optimizations, arithmetic, nodynamic_equiv)
2736      entry getoff(stp, symname, numchars, bitoffset)
2923      entry intrterm(xx)
```

INTR2

```
 47      entry line2reg(previous, linenum)
153      entry file2reg(previous, fname)
234      entry rvalline(dummy)
274      entry rvalfile(dummy)
313      entry makelabl(name, numchars)
364      entry bbreak(previous)
422      entry makext(name, numchars)
468      entry localvar(name, numchars, datatype, elements, bytesize, static)
561      entry namecom2(name, numchars, bitsize, common_type)
623      entry comnvar(name, numchars, datatype, elements, bytesize,
624     *              common_index, bitoffset)
727      entry inlfname(name, numchars        )
778      entry sple(previous, dagptr, linenum, pletype)
835      entry litval(binval, datatype)
877      entry strcon(stringptr, numchars)
970      entry asmcode(previous, linenum, stringptr1, numchars)
1084     entry sdage(operation, xshape, datatype, op1, op2, op3, namestp, xdbits)
1160     entry bextract(dagptr1, bytesize, dagptr2, justify, sign_extend)
1267     entry binsert(dagptr1, dagptr2, bytesize, dagptr3)
1338     entry dbinsert(dagptr1, dagptr2, bytesize_dagptr, dagptr3)
1412     entry byteoff(dagptr1)
1464     entry wdoffset(dagptr1)
1520     entry bitsb4(dagptr1)
1568     entry fbit(dagptr1, dagptr2)
1648     entry fbitaddr(dagptr1, dagptr2)
1728     entry cardinal(dagptr)
1782     entry kilvars(previous)
1935     entry getstp(name, numchars)
```

INTR3

```
 38     entry rval(name, numchars, level, bitoffset,          varmode,
 39    *          offdagp)
280     entry rvalvec(name, numchars, level, bitoffset,          varmode,
281    *              offdagp, lengthdagp)
455     entry rvalbv(name, numchars, level, bitoffset,          varmode,
456    *             bitoffdagp, lengthdagp)
643     entry lval(name, numchars, level, bitoffset,          varmode,
644    *          offdagp)
908     entry lvalvec(name, numchars, level, bitoffset,          varmode,
909    *              offdagp, lengthdagp)
1085    entry lvalbv(name, numchars, level, bitoffset,          varmode,
1086   *             bitoffdagp, lengthdagp)
1276    entry address(name, numchars, level, bitoffset,          varmode,
1277   *              offdagp)
1512    entry addrext(name, numchars)
```

INTR4

```
  38    entry rvalt(offdagp    )
  90    entry rvaltvec(offdagp,             lengthdagp)
 149    entry rvaltbv(bitoffdagp,            lengthdagp)
 279    entry lvalt(offdagp    )
 335    entry lvaltvec(offdagp,             lengthdagp)
 392    entry lvaltbv(bitoffdagp,            lengthdagp)
 503    entry addrt(offdagp          )
 557    entry rvali(dagptr1, offdagp          )
 633    entry rvalivec(dagptr1, offdagp,           lengthdagp)
 687    entry rvalibv(dagptr1, bitoffdagp,   lengthdagp)
 745    entry lvali(dagptr1, offdagp          )
 826    entry lvalivec(dagptr1, offdagp,           lengthdagp)
 880    entry lvalibv(dagptr1, bitoffdagp,   lengthdagp)
 940    entry addri(dagptr1, offdagp          )
1002    entry rvalvt(dagptr, offdagp          )
1077    entry addrvt(dagptr)
1148    entry eqvec(dagptr1, dagptr2)
1215    entry neqvec(dagptr1, dagptr2)
```

INTR5

```
 55      entry fmlparm(name, numchars, datatype, formalmode, bytesize, parmpos)
362      entry ffmlparm(bytesize, datatype, parmpos)
506      entry endspcal(name, numchars,       sprgtype, inline, blockbreak,
507     *                    genericflag)
708      entry endspdag(dagptr)
838      entry intrnsc(name, numchars)
956      entry endintrn(name, numchars)
1133     entry expparm(dagptr, actualmode)
1186     entry rtn(previous, dagptr, xxline, suppress_ple)
1450     entry callnew(dagptr1)
1511     entry calldisp(dagptr1, dagptr2)
1579     entry callmzln(previous)
1658     entry rtclock(dummy)
```

APPENDIX D

HYBLIB - Run Time Routines for Loading All Hybrid Compiled Programs

File minit:
Minit contains the routines for main program initialization and termination.
maininit - main program initialization
mainexit - main program termination

File mzline:
Mzline contains the gateway to the NLTSS debug package
mzline - very special entry to the NLTSS debug package

File new:
New contains all the routines for heap management and heap tracing
dispose - return a block to the heap manager
htrace - heap tracing routine
new - get a block from the heap manager
tracedst - set heap tracing destination
tracehp - enable (disable) heap tracing

File rterr:
Rterr contains the run time error routines and message formatting routines.
mzdunerr - discriminated union error routine
mzmsg1 - internal formatting routine
mzmsg2 - internal formatting routine
mznilerr - nil pointer reference error routine
mzovflo1 - internal debugging error routine
mzovflo2 - internal debugging error routine
mzreterr - exit function without explicit return error routine
mzrngerr - range (bounds) error routine

APPENDIX E

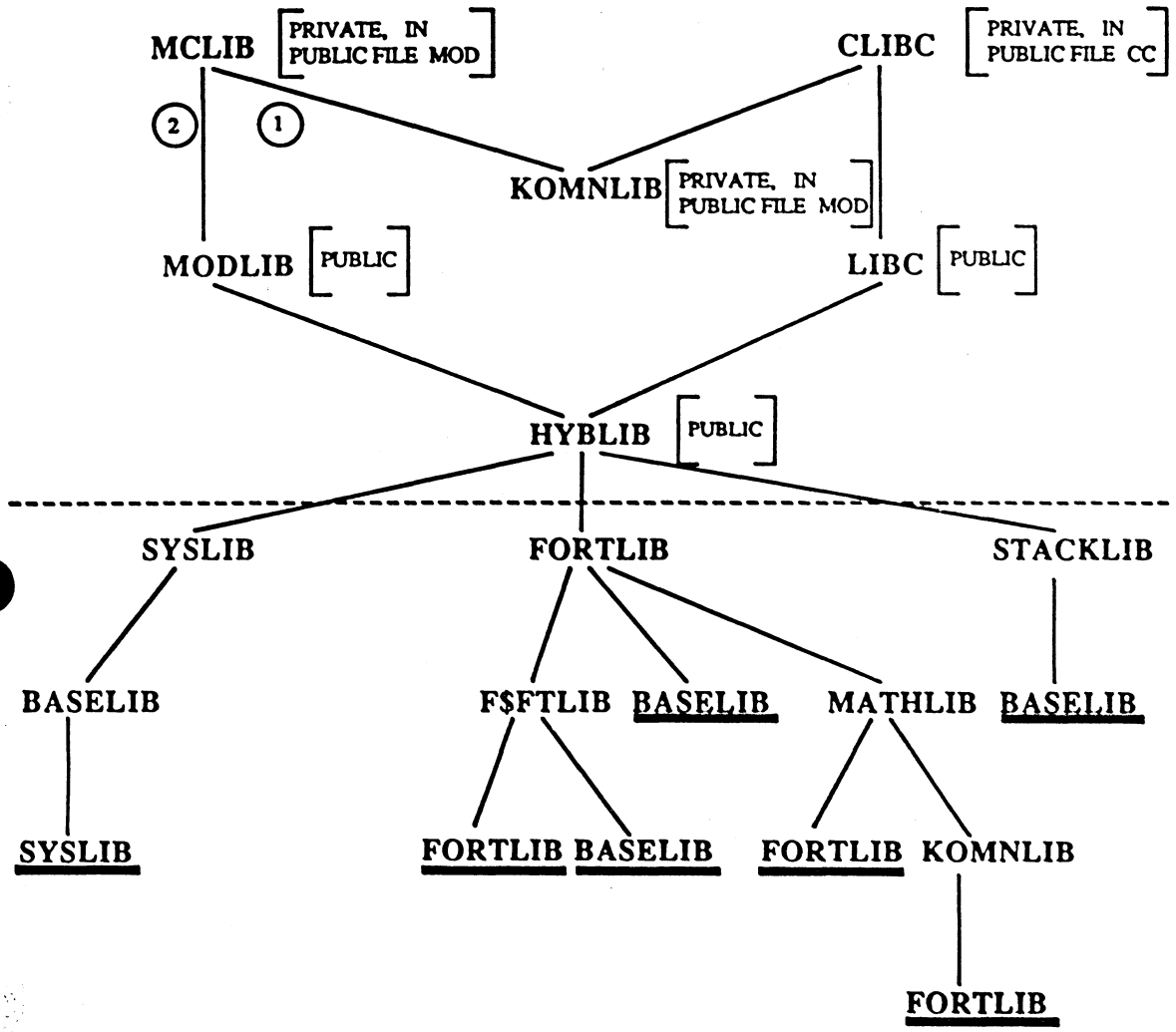
KOMNLIB - Additional Run Time Routines for Loading the Hybrid Compilers Themselves

File mzget:
Mzget contains the routines which manage the restricted heap currently used by the hybrid compilers. This whole area will be redesigned when we switch to dynamic table management in the Civic backend.

hpstats	- heap usage statistics
mzget	- mzget for the system memory manager
sethprms	- set heap expansion parameters
stopheap	- stop restricted heap from expanding

APPENDIX F

HYBRID LIBRARY DEPENDENCIES (5-4-87)



The order of searching supporting libraries is always left to right except where noted otherwise (by a number in a circle). For example, after searching mclib, the loader searches komnlib before modlib.

APPENDIX G

Where the Files are Stored
(public files and xport directories)

Public files:

HYBLIB - a build library containing the routines described in APPENDIX I
MOD - a lib file containing KOMNLIB
(komnlib contains the routines described in APPENDIX J)

Xport directory structure:

.hybrid[sor900v bin900v]
.hybrid:binsor
.hybrid:libs

In the above text, sor900v and bin900v refer to the current working files for the hybrid interface and civic backend. Sor900v is a lib file that contains the source files, and bin900v is a build library which contains the corresponding binaries. When a version becomes frozen, it is moved to .hybrid:binsor for permanent storage.

The *clones* of bin900v, that is, the binary versions for various other combinations of machines, operating systems, and external symbol table formats, are kept in mnemonically named subdirectories of .hybrid:binsor:clones. For example, all the clones of bin900v are kept in directory

.hybrid:binsor:clones:900v.

See the section, "How to Load Hybrid Compilers", for a summary of the clones and clone naming conventions. The libraries associated with the hybrid compiler writing system are explained on the following pages.

Hybrid Library Directory Structure

```
                    :mkmclib
.hybrid:libs:mclib:n[ cs<date>   cb<date>  ]
                -   :c[ -        " -       ]
                    :o[          "         ]


                    :mkclibc
.hybrid:libs:clibc:n[ ls<date>   lb<date>  ]
                -   :c[ -        " -       ]
                    :o[          "         ]


                     :mkcomnlib
.hybrid:libs:komnlib:n[ ks<date>   kb<date>  ]
                -    :c[ -         " -       ]
                     :o[           "         ]


                    :mkmodlib
.hybrid:libs:modlib:n[ ms<date>   mb<date>  ]
                -   :c[ -         " -       ]
                    :o[           "         ]


                    :mkhyblib
.hybrid:libs:hyblib:n[ hs<date>   hb<date> ]
                -   :c[ -         " -       ]
                    :o[           "         ]
```

The subdirectories n, c, o have the meanings:

n - new, pertains to new software currently in development
c - current, pertains to software currently in public
c - old, pertains to software that has previously been in public

The actual file names follow these conventions:

The FIRST character of the file name indicates to which library it belongs.

c - mclib (For loading **Model hybrid compilers** only).

l - clibc (For loading **C hybrid compilers** only).

k - komnlib (For loading **all hybrid compilers).**

m - modlib (For loading programs compiled with a Model hybrid compiler)

h - hyblib (For loading programs compiled with any hybrid compiler)

The SECOND character of the file name indicates source or binary.
s - A LIB file containing SOURCE files.
b - A BUILD library containing the corresponding binary.

EXAMPLE:
The name cs851231 is an encoding of the following information:
c - mclib
s - source

The files which start with mk are simple cosmos files which contain the the compiler command lines for the corresponding source files.

The actual file names follow these conventions:

APPENDIX H

Interface Routine Prologs

ACTLPARM

Actual parm. Add an actual parm to the current parm list.

Calling sequence:
call actlparm(dagptr)

Input:
dagptr - (pexp.shl.pntrsiz) + dag pointer for the next actual parm.
The dag must be a scalar dag. It may NOT be a vector dag.

Output:
None.

Operation:
Dagptr is added to the parm list for the currently active invocation of spcal.

Actlparm is initialized by spcal and uses the globals:
firstparm, lastparm, and nextparm.

ADDRESS

Address of a scalar. Used for other than LHS of assignment.

Calling sequence:

dagp1=address(name, numchars, level, bitoffset, varmode,
offdagp)

Input:

name - an array containing ascii for the name of the variable for which we are building a left subgraph.
numchars - the number of non-blank chars in the name.
level - undefined for common variables. (Code as zero for efficiency).
- static nesting level of the declaration of the variable for which we are building a left subgraph.
bitoffset - for LOCAL AND COMMON VARIABLES: undefined
- for NON-LOCAL VARIABLES: bit offset within appropriate AR for start of the non-local object. This value was returned by getoff during the compilation of a previous subprogram (or the main program).

varmode - What kind of variable is this?
notparm - non-parm
valparm - value formal parm
refparm - reference formal parm
callee_copy_parm - the CALLEE made of copy of the parm in its own AR. Only used for certain multi-word objects.
common_var - common variable
offdagp - (offset dag pointer):
zero - no offset required
nonzero - a dag pointer representing a WORD offset into the object specified by the prior parameters. This allows us to "index" into records, arrays, arrays of records, etc..

Output:

name - when numchars is not a multiple of 8, name will be space filled out to a multiple of 8 chars.
The function value is (pexp.shl.pntrsiz) + dag pointer to a dag subgraph representing the left value of the variable specified by name. This left value cannot be used on the left side of an assignment statement. Use lval (or its successors if some are created after I write this) in that case. Address is only used when the left value (address) of a declared object (not a copy of the object) is to be placed into an actual parm list. Therefore the paramb bit gets set.

Operation:
The subgraphs produced by address are frequently occuring graphs. This routine returns a dag pointer to the appropriate kind of subgraph for the variable named in input parm, name. When offdagp is nonzero, then we are referring to a word within the object.

In address there are 6 basic types of scalar variable references. So in general, six different dag subgraphs are required to reference them. However, the dag subgraphs required for non-parm variables and value (formal) parms are identical (one for locals, another for non-locals) Hence, internally, address generates only 4 different basic subgraphs. They are numbered 1 - 4 in the table below.
The dags become more complicated when offdagp is nonzero.

	level	
	level = clevel local	level < clevel non-local
varmode		
1 - non-parm	1	3
2 - value formal	1	3
3 - reference formal	2	4
4 - callee makes copy of multi-word formal	1	3
5 - common variable		

ADDREXT

Address of an external

Calling sequence:
dagp1=addrext(name, numchars)

Input:
name - an array containing ascii for the name of the external.
numchars - the number of non-blank characters in the name.

Output:
name - When numchars is not a multiple of 8, name will be space filled out to a multiple of 8 chars.
The function value is (pexp.shl.pntrsiz) + a dag pointer for the PARCEL addr of the external.

Operation:
A symbol table for the external is built and an opaque dag pointing to that symbol table entry is also built.

ADDRI

Address (indirect). Address of an object pointed to by a pointer variable.
For OTHER than LHS of assignment statement.

Calling sequence:
dagp1=addri(dagptr1, offdagp)

Input:
dagptr1 - (pexp.shl.pntrsiz) + a dag pointer representing the VALUE of a POINTER variable (this value is actually the address of the object pointed to).
offdagp - 0 or (pexp.shl.pntrsiz) + a dag pointer representing a word offset into the object pointed to.

Output:
The function value is (pexp.shl.pntrsiz) + a dag pointer representing the ADDRESS of the WORD at:
base address of object pointed to + word offset (from offdagp).
Note: This dag cannot be used on the LHS of an assignment statement. Use rvali in that case. Addressi is only used when an address is needed for a purpose other than assignment, for example, when an address is needed for a parm list.

Operation:
Suppose you have a dag for the VALUE of a pointer variable, but what you really want is the address of some word relative to the beginning of the object pointed to. Addri creates a dag which represents the desired address. This is a common operation when dereferencing pointer variables.

ADDRT

Address of Temporary. Address of a word in the hybrid temporary area. For OTHER than LHS of assignment statement.

Calling sequence:
dagp1=addrt(offdagp)

Input:
offdagp - 0 or (pexp.shl.pntrsiz) + a dag pointer representing a word offset expression relative to the start of the hybrid temporary storage area.

Output:
The function value is (pexp.shl.pntrsiz) + a dag pointer representing the address of the word in the hybrid temporary storage area at the offset represented by offdagp. This dag can only be used in places OTHER than the LHS of an assignment statement.

Operation:
The result dag is computed based on the input parms.

ADDRVT

Address of a vector temporary
SIDE EFFECTS! SIDE EFFECTS! TEMPORARY VECTOR BUILT!!!

Calling sequence:
dagp1=addrvt(dagptr)

Input:
dagptr - (pexp.shl.pntrsiz) + a dag pointer representing a vector (word or bit) which is THE RESULT OF AN EXPRESSION. Specifically note that this routine will NOT ACCEPT an OPVAL as input.

Output:
The function value is (pexp.shl.pntrsiz) + a dag pointer representing the word address of the start of the vector temporary which now contains the vector represented by dagptr.

Operation:
This routine takes a vector that was created as the result of a vector operation (represented by dagptr) and stores it in a temporary location (probably the vector temporary area). The function value is as described above.

Note: This routine performs an error stop if dagptr is an opval dag, so don't try it. It also performs an error stop if dagptr is not a vector dag.

Note: If this routine is called, remember to update the vector temporary usage information which is passed to subroutine backend in parameter temp2siz.

ASMCODE

Assembler Code

Calling sequence:
lastple=asmcode(previous, linenum, stringptr1, numchars)

Input:
previous - index to the previous ple
linenum - the line number to store into the ple
stringptr1 - a variable containing the address of an array which contains the ascii characters of the specially formatted string which represents the inline assembly code. See below for the required format of this string. The last word of this array must be padded with spaces.
numchars - the number of characters in the string (in general this is not a multiple of 8, but the last word must be padded with spaces).

Output:
The function value is the index of the newly created ple. This ple points to an opfcall dag for the inline function a__emble
(which how we represent inline assembly code in the civic dag language).

Operation:
Stosym (not stosymn) is used to build a literal table entry for the string constant representing the inline assembly code, and then we create an opfcall dag where:
dop1 points at a__emble
dop2 points at an opval for the string.

Notes:
Format of the string representing the inline cal:
. #1f is the ascii char us .
#01 is the ascii char soh .
. The pair of ascii chars us soh signals the beginning of each line of cal.
. Each field of a cal instruction is separated by a SINGLE space. Please note, this says a SINGLE space!!!
. If the line starts with a label, then the ascii chars for the label immediately follow the us soh pair.
If the line does not start with a label, then there is a SINGLE space between the us soh pair and the start of the first cal field.
. The entire string ends with another us soh pair. This last us soh pair is included in numchars (but no chars beyond these are included). This count must be exact.

Additional notes:
It is the responsibility of the front end to detect cal labels and to

call the interface routine, makelabl, for every detected label.
These labels should not conflict with any other user defined labels or
compiler generated labels.

BACKEND

Execute the Civic backend

Calling sequence:

call backend(startline, numlines, temp1siz, temp2siz, highfast_temp, optimizations, arithmetic, nodynamic_equiv)

Input:

startline- the starting line number in the input file where the source listing should start for this subprogram (or main program).

numlines - the number of lines to list for this subprogram (or main program.

temp1siz - the number of words needed for the hybrid temporary storage area

temp2siz - the number of words needed for the vector temporary storage area. If this value is zero, then initialization of the local variable, %iq8qdsp, does not occur during procedure initialization.

highfast_temp - the highest numbered "fast" temp actually used for this subprogram (values go from 0 to (maxfast_temps - 1)).

optimizations - 0 - use the optimizations specified in the call to glinit

nz - we are overiding the optimization levels specified in the call to glinit for this subprogram only.

VALUES: Any mixture of ascii chars b,c,d,e,f,g,i left justified space or null filled.

Each letter specifies a particular optimization.

If only one letter is specified, then it and all lower optimizations are selected.

arithmetic - 0 - use the arithmetic specified in the call to glinit.

nz - we are overriding the arithmetic specified in the call to glinit for this subprogram ONLY.

VALUES: "s", "l", or "r". Same meaning as civic.

nodynamic_equiv - 0 - use the vectorization as specified in the call to glinit.

nz - This has approximately the same effect as the civic command line parameter v=a but for this subprogram ONLY. See glinit or the Irltran manual sec. 3.4.2, p. 122 for details of v=a. (The difference in meaning is that this parameter does not change the sense of v=t as specified in the call to glinit).

It is assumed that the dags, symbol table entries, literal table entries, and auxiliary data structures for the next subprogram to be processed have been built.

Output:

Nothing is passed back to the calling program.

Operation:

The optimization, code generation, and binary and listing output phases of the Civic backend are performed. After backend returns, the hybrid interface can be reinitialized for another subprogram (procinit), another set of dags can be built, and backend can be called again.

BBREAK

Break a block.

Calling sequence:
lastple=bbreak(previous)

Input:
previous - pointer to the previous ple.

Output:
The function value is a pointer to a newly create ple for a used label which is how we break the block.

Operation:
A ple which points to a symbol table entry for a used label is created and chained after the ple pointed to by input parm, previous. This routine is used whenever the front end needs to break a block for whatever reason. The original reason we created this routine is because the civic backend is not designed to handle dynamic name equivalencing (aliasing) when doing optimization and sometimes it is necessary for the front end to break blocks in order to get correct code (especially at o=g).

Note:
The name of the label created is always -bbreak. Thus ple's for this one label can occur all over your code. The backend doesn't seem to mind this at all.

Calling sequence:

BEXTRACT

Byte Extract. Extract a specified byte from a specified word. Bextract is designed to extract bytes LESS than a word size, only.

Calling sequence:

dagp1=bextract(dagptr1, bytesize, dagptr2, justify, sign_extend)

Input:

dagptr1 - (pexp.shl.pntrsiz) + a dag pointer which represents the value of the full WORD from which we want to extract a byte (source value).
Typically computed by rval, rvalt, rvali.

bytesize - the size in bits of the byte to be extracted from the source value.

dagptr2 - (pexp.shl.pntrsiz) + a dag pointer which represents the number of bits in the source value preceding the byte that we want to extract. For example, if the byte is left justified in the source value, the the value of this dag expression is 0.
We need a dag here because in general, this value must be computed at run-time (due to the indexing of arrays with elements less than word size).

justify - 0 - right justify the extracted byte in the result value (0 fill).
1 - left justify the extracted byte in the result value (0 fill).

sign_extend - 0, don't do sign extension
1, do sign extension
The special case of sign_extend = 1 and justify = 1 is treated as an error.

Output:

The function value is (pexp.shl.pntrsiz) + a dag pointer to the opextract dag which represents the value of the extracted byte (right justified). If left justification is requested, then the dag pointer is to the opshl dag which represents the value of the extracted byte (left justified).

Operation:

An opextract dag to perform the byte extract from a word is built by appropriately combining the input parms. This gives a right justified result. If left justification is requested, then an opshl dag is added.

A pair of opshl's are generated if sign extension is specified.

BINSERT

Byte Insert. Insert a specified byte, from a specified source WORD, into another specified WORD. Binsert is designed to insert bytes LESS than a word size, only.

Calling sequence:
dagp1=binsert(dagptr1, dagptr2, bytesize, dagptr3)

Input:
dagptr1 - (pexp.shl.pntrsiz) + a dag pointer which represents the Right value of the full WORD into which we want to insert a byte (target value).
This is typically computed by rval, rvalt, rvali.
dagptr2 - (pexp.shl.pntrsiz) + a dag pointer which represents the value of where to get the new bits to insert (the rightmost bits are used).
bytesize - the size in bits of the byte to be overstored in the target value
dagptr3 - (pexp.shl.pntrsiz) + a dag pointer which represents the number of bits in the source value preceding the byte that we want to extract. For example, if the byte is left justified in the source value, then the value of this dag expression is 0).
We need a dag here because in general, this value must be computed at run-time (due to the indexing of arrays with elements less than word size.

Output:
The function value is (pexp.shl.pntrsiz) + a dag pointer representing the new value with the specified bits inserted properly.

Operation:
A dag to achieve the insert is built.

BITSB4

BITS before (B4). Bit offset WITHIN a word.

Calling sequence:
dagp1=bitsb4(dagptr1)

Input:
dagptr1 - (pexp.shl.pntrsiz) + a dag pointer representing a bit address or a bit offset

Output:
Let offset1 be the offset computed by subprogram wdoffset with input dagptr1. The function value of bitsb4 is (pexp.shl.pntrsiz) + a dag pointer representing the expression:

(input bit offset) - (word size)*offset1 .

This is a bit offset WITHIN the word at offset1. For example, if the object specified by the input bit offset (dagptr1) is left justified in the word, then bitsb4 represents the value 0.

Operation:
The result dag is built based on the input parm.

BYTEOFF

Byte offset. Map bit offset to byte offset (8 bits per byte).

Calling sequence:
dagp1=byteoff(dagptr1)

Input:
dagptr1 - (pexp.shl.pntrsiz) + a dag pointer representing a bit address
or a bit offset

Output:
The function value is (pexp.shl.pntrsiz) + a dag pointer representing the byte offset which contains the input bit offset.
For example:
bit offsets 0 - 7 map to byte offset 0.
bit offsets 8 - 15 map to byte offset 1.
etc.

Operation:
This routine maps a bit offset dag into the corresponding byte offset dag.

Note:
The following code assumes a byte addressing space of 27 bits and a word size of 64 bits.

CALLDISP

Call dispose, to deallocate a heap block.

Calling sequence:
dagp1=calldisp(dagptr1, dagptr2)

Input:
dagptr1 - (pexp.shl.pntrsiz) + a dag pointer for the address of the heap block to be deallocated. Dagptr1 is typically computed by calling rval for a pointer variable.
dagptr2 - (pexp.shl.pntrsiz) + a dag pointer for the size (in words) of the heap block to be deallocated.

Output:
The function value is (pexp.shl.pntrsiz) + a dag pointer for the function call to dispose.

At run-time, dispose always returns the value, nil (74000000b). (#f00000) One can then store this value into the pointer variable which used to contain the address of the heap block which has just been deallocated. In the alternative, one could hang this dag directly off a ple, but this seems less useful. (Opfcalls for subroutines and functions can legally hang directly off ple's).

Operation:
Calldisp builds the appropriate dags to call the function, dispose.

CALLMZLN

Call mzline

Calling sequence:
lastple=callmzln(previous)

Input:
previous - The ple after which the ple created by this routine should be chained.

Output:
The function value is the index to a texp ple which calls mzline as a generic subprogram with no parameters.

Operation:
callmzln creates a ple which calls the routine mzline as described below. This ple is chained after the ple specified by the input parm, previous. The function return value points to this new ple.

It is expected that a cal routine called mzline will exist in modlib, libc, etc.

callmzln was created to be compatible with the model debug package which allows dynamic setting of breakpoints immediately before the generated code for each source line.
If mzline is to be called, it should be called immediately AFTER setting the line number register (line2reg) for each source line.

Although this debugging approach was designed with model in mind, it is probably useful with codes produced with other hybridized compilers.

Mzline must be carefully coded in cal to achieve the desired effect for a given language.

Also note, calls to mzline break basic blocks. Therefore, if you take one source file and compile it once with calls to mzline enabled, and compile it again with calls to mzline disabled, the resulting executables will reflect different optimizations. Thus it is possible for one of the codes to malfunction (probably the one without calls to mzline because it has undergone more extensive optimization due to larger basic blocks) while the other code functions correctly. Awareness of this property is important when attempting to use the mzline feature to help debug a code.

CALLNEW

Call new, to allocate a heap block.

Calling sequence:
dagp1=callnew(dagptr1)

Input:
dagptr1 - (pexp.shl.pntrsiz) + a dag pointer for the value of the size (in words) of the requested heap block. Dagptr1 is typically computed by calling rval for a pointer variable.

Output:
The function value is (pexp.shl.pntrsiz) + a dag pointer for the function call to new.
At run-time, new returns the following values:
If the request was satisfied: the address of the new block.
o.w.: nil (74000000b) (#f00000)

Operation:
callnew builds the appropriate dags to call the function, new.

CARDINAL

Cardinal (cardinality). Sum the number of 1 bits in a bit vector.

Calling sequence:
dagp1=cardinal(dagptr)

Input:
dagptr - (pexp.shl.pntrsiz) + a dag pointer for a bit vector.

Output:
The function value is (pexp.shl.pntrsiz) + a dag pointer representing the number of 1 bits in the input bit vector.

Operation:
Dags to sum the 1 bits in the input bit vector are built. Inline function q8sum is used.

COMNVAR

Declare a common variable

Calling sequence:

stp1=comnvar(name, numchars, datatype, elements, bytesize,
common_index, bitoffset)

Input:

name - an array containing ascii for the name of the common variable
for which we are building a symbol table entry.
numchars - the number of non-blank chars in the name.
datatype - stored into dt of the found or created symbol table entry
elements - 0 if this object is NOT an array.
The total number elements if the object IS an array.
bytesize - bytesize (when datatype = pbyte)
When DATATYPE is:
pbyte - bytesize is the size in bits of the object being
declared. (For an array, bytesize is the size in
bits of each element in the array).
o.w. - bytesize is undefined
common_index - the index returned by namecom2 for the common block which
contains this variable
bitoffset - the bitoffset into the common block where this variable begins
0 - means left justified in word 0
64 - means left justified in word 1
64*n - means left justified in word n
I think that non-multiples of 64 are just giving info to ddt
and that you should use a multiple of 64 if you are going to
reference this symbol table entry with dags.

Output:

name - when numchars is not a multiple of 8, name will be space
filled out to a multiple of 8 chars.
The function value is the index to the found or created symbol table
entry. Note, pvar is not included in this value so it is not suitable
for storing into the DOPn field of a dag.

Operation:

This routine finds or creates a symbol table entry for the given name
and sets it up properly for a common variable. It must be called
AFTER the call to namecom2 for the corresponding common block.

Note:

comnvar may be called a second time for a given name if you don't know
the values for element, bytesize and/or bitoffset on the first call.
(In that case, just use 0). All the other parms should be the same on
both calls. Two calls for the same name may be necessary due to C

initializers, for example, where we don't have the size information at the time we must build the symbol table entry.

DATALADR

Data load the address (load time computable) of some object plus (perhaps) a a constant offset (compile time computable).

Calling sequence:
call datalval(symptr1, wdoff1, bytesize, bitsbefore, symptr2, wdoff2)

Input:
symptr1, wdoff1, bytesize, and bitsbefore describe the word(or partial word) which is to be data loaded.

symptr1 - symbol table pointer (without pvar) for the object containing the word or partial word to be data loaded. This is probably the return value from a call to getstp.
wdoff1 - a 0 origin wd offset into object specified by symptr1 specifying which word is to be data loaded (or which word contains the byte to be data loaded).
bytesize - zero - load the entire word specified by symptr1 and wdoff1
- maybe I should change this to 64 (but maybe that's too machine dependent).
nz - load only a partial word, this is the size in bits of the byte to be loaded.
bitsbefore - when bytesize = 0, this parm is undefined
o.w., this specifies where the byte of interest starts in the word. bitsbefore says how many bits precede this byte in the word. For example, if bitsbefore = 0, then this byte is left justified in the word.
symptr2 - symbol table pointer (without pvar) for the object whose addr is to be data loaded into the word or partial word specified by the preceding parms. This is probably the return value from a call to getstp.
wdoff2 - add this value to the addr specified by symptr2 before doing the data load. Note, this computation must be done by the loader because the compiler can't know what the value of the desired addr is. The loader decides that.

Output:
Some fields are set in symptr1 (see below).

Operation:
The data structure to specify the desired data load is built via a call to datldvar. The following fields are set in the civic symbol table entry pointed to by symptr1:
datlb=1

A brief review of the rb field (for rb>2) in a symbol table entry:
The symbol table entry for each EXTERNAL object referenced by a program or

subprogram
  . external subprogram
  . external entry point
  . block data subprogram
receives a unique rb value (rb>2). However, this value is set AFTER the dags have been built. Therefore, at this point, rb fields for those objects are zero.

However, each variable in a common block does have a rb>2 at this point. (The rb field identifies which common block contains this variable).

DATALVAL

Data load a value (compile time computable)

Calling sequence:
call datalval(symptr1, wdoff1, bytesize, bitsbefore, value)

Input:
symptr1, wdoff1, bytesize, and bitsbefore describe the word(or partial word) which is to be data loaded.

symptr1 - symbol table pointer (without pvar) for the object containing the word or partial word to be data loaded. This is probably the return value from a call to getstp.
wdoff1 - a 0 origin wd offset into object specified by symptr1 specifying which word is to be data loaded (or which word contains the byte to be data loaded).
bytesize - zero - load the entire word specified by symptr1 and wdoff1
- maybe I should change this to 64 (but maybe that's too machine dependent).
nz - load only a partial word, this is the size in bits of the byte to be loaded.
bitsbefore - when bytesize = 0, this parm is undefined
o.w., this specifies where the byte of interest starts in the word. bitsbefore says how many bits precede this byte in the word. For example, if bitsbefore = 0, then this byte is left justified in the word.
value - the value to data load

Output:
Some fields are set in symptr1 (see below).

Operation:
The data structure to specify the desired data load is built via a call to datldvar. The following fields are set in the civic symbol table entry pointed to by symptr1:
datlb=1

A brief review of the rb field (for rb>2) in a symbol table entry:
The symbol table entry for each EXTERNAL object referenced by a program or subprogram
- external subprogram
- external entry point
- block data subprogram

receives a unique rb value (rb>2). However, this value is set AFTER the dags have been built. Therefore, at this point, rb fields for those objects are zero.

However, each variable in a common block does have a rb>2 at this point. (The rb field identifies which common block contains this variable).

However, each variable in a common block

DBINSERT

Dag, Byte Insert. Insert a specified byte, from a specified source WORD, into another specified WORD. Dbinsert is designed to insert bytes LESS than a word size, only.

Calling sequence:
dagp1=dbinsert(dagptr1, dagptr2, bytesize_dagptr, dagptr3)

Input:
dagptr1 - (pexp.shl.pntrsiz) + a dag pointer which represents the Right value of the full WORD into which we want to insert a byte (target value).
This is typically computed by rval, rvalt, rvali.
dagptr2 - (pexp.shl.pntrsiz) + a dag pointer which represents the value of where to get the new bits to insert (the rightmost bits are used).
bytesize_dagptr - (pexp.shl.pntrsiz) + a dag pointer representing the size (in bits) of the byte to be overstored in the target value.
dagptr3 - (pexp.shl.pntrsiz) + a dag pointer which represents the number of bits in the source value preceding the byte that we want to extract. For example, if the byte is left justified in the source value, then the value of this dag expression is 0).
We need a dag here because in general, this value must be computed at run-time (due to the indexing of arrays with elements less than word size.

Output:
The function value is (pexp.shl.pntrsiz) + a dag pointer representing the new value with the specified bits inserted properly.

Operation:
A dag to achieve the insert is built.

Note:
Dbinsert differs from binsert only in that the bytesize parm is a Dag pointer rather than a constant.

ENDINTRN

End intrinsic function call

Calling sequence:
dagp1=endintrn(name, numchars)

Input:
name - one word containing the source language name of an Irltran intrinsic function (up to 8 chars, left-justified-space- or zero-filled).
numchars - the number of nonblank chars in name

Output:
name - When numchars is not 8, name will be space filled out to 8 chars.
The function value is (pexp.shl.pntrsiz) + dag pointer to an opfcall or opinlfunc dag representing the intrinsic function.Note that in general you are NOT calling an external function with the name in name. In the dags, all the actual parms recorded by actlparm are properly chained (with opcomma dags if necessary) under the opfcall or opinlfunc dag.

Operation:
Internally this routine works very much like endspcal so read the the description there. The major difference is that this routine uses its own internal table to figure out exactly how to build the dags rather than relying on a bunch of input parms. Also, usually (always?) the name supplied in name does not actually wind up in the symbol tables. This name is mapped to some other name that the civic backend end knows how to deal with in special ways (the essence of an intrinsic function is that the compiler has prior knowledge of its properties).

Something I noticed:
For exp and a few other intrinsic functions:
The civic frontend builds opinlfunc-opaddr-exp%, but extb is NOT set in exp% (at any optimization level).
If you run at o=i, a symbol table entry for %exp% appears and the entry for exp% remains a non-external. However, if you run without o=i, then by the time the compiler is finished, someone after the front end sets the extb bit in exp% and %exp% does not appear.

My assumption is that %exp% must be a vector version of exp% and we don't want to set extb when unnecessary and therefore load unneeded modules.

ENDSPCAL

End subprogram call. (Also handles "calls" to civic inline functions).

Calling sequence:

```
dagp1=endspcal(name, numchars,      sprgtype, inline, blockbreak,
*              genericflag)
```

Input:

name - an array containing ascii for the name of the subprogram
numchars - the number of non-blank characters in the name.

sprgtype is currently ignored and is now a candidate for deletion.
sprgtype - subprogram type
2 - subroutine (subtype)
3 - function (functype)

inline - nonzero - we are generating dags to "call" a civic inline function. In this case, the name is assumed to exist in the civic code generator's list of inline function names.
zero - (normal case), we will call an external subroutine with regular subroutine linkage.
blockbreak - nonzero - (normal case) this subprogram call breaks a block (i.e., an opfcall dag is generated).
zero - this subprogram call will NOT break a block (i.e., an opinlfunc dag is generated).
genericflag - 1 - this is a generic subprogram (the parms are passed by value in registers).
0 - normal subprogram.

Output:

name - when numchars is not a multiple of 8, name will be space filled out to a multiple of 8 characters.
The function value is (pexp.shl.pntrsiz) + dag pointer to an opfcall or opinlfunc
dag representing the subprogram call. All the actual parms recorded by actlparm are proprerly chained (with opcomma dags if necessary) under the opfcall dag,.

Operation:

The output dag as described above is computed. Endspcal pops the stack for subprogram control variables so a subsequent call to actlparm will add a parm to the previous parm list (if any).

Discussion:

Distinction Between Opinlfunc and Opfcall Dags

An opinlfunc dag does not indicate an in-line function. An opinlfunc dag can

call any subroutine or function just like an opfcall.

Distinction:

opinlfunc - NEVER breaks a basic block
opfcall - ALWAYS breaks a basic block

Signalling in-line functions:

If dop1 of an OPINLFUNC or OPFCALL dag points directly at a symbol table entry for a subprogram call (not at an opaddr which points at the symbol table entry then we are telling the code generator that we expect in-line code to be generated. If the code generator has that "subprogram" name in its internal table of inline names, then in-line code will be generated. Otherwise, there will be a code generator error.

All four combinations are possible.

Examples:

opinlfunc for MOST inline functions.
opinlfunc for external subprogram, new. (This sort of "safe" subprogram is referred to as a "library function"

Break a block:

opfcall for inline code to call the clock.
opfcall for MOST external subprograms.

ENDSPDAG

End subprogram call, dag (parcel addr of the subprogram is expressed by a dag)

Calling sequence:
dagp1=endspdag(dagptr)

Input:
dagptr - (pexp.shl.pntrsiz) + a dag pointer for an expr which evaluates to the parcel addr of a subprogram which is to be called. Most often I would expect the expr to merely be the value of a formal parm or "pointer" variable which contains this parcel addr. However, any expression will do.

Output:
The function value is (pexp.shl.pntrsiz) + dag pointer to an opfcall dag representing the subprogram call. All the actual parms recorded by actlparm are proprerly chained (with opcomma dags if necessary) under the opfcall dag,.

Operation:
The output dag as described above is computed. Endspdag pops the stack for subprogram control variables so a subsequent call to actlparm will add a parm to the previous parm list (if any).

EQVEC

Equal (vector). Test equal between two MULTI-WORD objects. Multi-word operations are implemented using vector dags.
This also works for bit vectors.

Calling sequence:
dagp1=eqvec(dagptr1, dagptr2)

Input:
dagptr1 - a vector right value dag for some multi-word object.
typically computed by rvalvec, rvaltvec, rvalivec
dagptr2 - same as dagptr1
dagptr1 and dagptr2 can also be bit vectors.

Output:
The function value is (pexp.shl.pntrsiz) + a dag representing an integer whose value is:
0 - (FALSE) When the two multi-word objects are NOT equal.
1 - (TRUE) When the two multi-word objects ARE equal.

Operation:
The result dag is built based on the input parms.

EXELINE

Execute Line (Set the text and size of the execute line)

Calling sequence:
call exeline(text, numwords)

Input:
text - an array containing the ascii text of the execute line to be printed at the end of the civic listing file. The last meaningful word of text MUST be SPACE filled.
numwords - The number of meaningful WORDS in this array. Thus the last meaningful word is text(numwords).

Output:
None

Operation:
The appropriate civic common variables are set so that the execute line will appear at the end of the civic listing file. Note that the code in civic routine execute (file rest) where the execute line is printed has some obscure formatting conventions which I discovered by by experimentation. So sometimes, if the text array contains blanks, some of them will be compressed out according to rules more complicated than I want to get into here. In any case, even when blanks are compressed out the execute line still looks pretty readable so I kept this routine simple rather than trying to print out the execute line EXACTLY as it appears in the variable, text.

EXPPARM

Expression parm. Create a dag for an expression to be used as an actual parm.

Calling sequence:
dagp1= expparm(dagptr, actualmode)

Input:
dagptr - (pexp.shl.pntrsiz) + a dag pointer to a dag representing the value of the expression to be added to the actual parameter list.
actualmode - What is the parameter passing mode of this actual?
valparm - value
refparm - reference

Output:
The function value is (pexp.shl.pntrsiz) + a dag pointer for an expression used as an actual parameter. Later, this dag will be chained under the appropriate opfcall dag by actlparm.

Operation:
Expparm builds an appropriate dag subgraph based on dagptr and actualmode

FBIT

First Bit

Calling Sequence:
dagp1 = fbit(dagptr1, dagptr2)

Input:
dagptr1 - (pexp.shl.pntrsiz) + a dag pointer representing a boolean array <= 64 bits long.
dagptr2 - (pexp.shl.pntrsiz) + a dag pointer representing the length of the boolean array.

Output:
The function value is (pexp.shl.pntrsiz) + a dag pointer representing the bit position where the first 1 bit was found in the boolean array. Bit positions are 0-origin, so if the leftmost bit in the boolean array is set, the result dag represents the value, zero. (Note, this value is equal to the number of leading zeroes in the boolean array). If all the bits are zero, the function value is the length of the boolean array.

Operation:
Dags to compute the desired value are built. A subroutine call to the generic stacklib routine, ivleadz, is generated using an opinlfunc dag.

FBITADDR

First Bit, Address.

Calling Sequence:

dagp1 = fbitaddr(dagprt1, dagptr2)

Input:

dagptr1 - (pexp.shl.pntrsiz) + a dag pointer representing the WORD ADDRESS of a boolean array (typically used for arrays > 64 bits but not necessarily so. This can be used for a short array also. However, one usually uses fbit for a short array because it's more convenient.
strictly necessary).

dagptr2 - (pexp.shl.pntrsiz) + a dag pointer representing the length of the boolean array.

Output:

The function value is (pexp.shl.pntrsiz) + a dag pointer representing the bit position where the first 1 bit was found in the boolean array. Bit positions are 0-origin, so if the leftmost bit in the boolean array is set, the result dag represents the value, zero. (Note, this value is equal to the number of leading zeroes in the boolean array). If all the bits are zero, the function value is the length of the boolean array.

Operation:

Dags to compute the desired value are built. A subroutine call to the generic stacklib routine, ivleadz, is generated using an opinlfunc dag.

FFMLPARM

Fake formal parm. Return a dag pointer (to a vector lval) for the fake formal parameter by means of which we return a MULTI-WORD function value.

Calling sequence:
dagp1=ffmlparm(bytesize, datatype, parmpos)

Input:
bytesize - The size in bits of the multi-word function return value.
When datatype .ne. pbit, a FULL WORD vector dag is returned. Enough words are specified to contain the number of bits in bytesize
When dat_ -q. pbit, a BIT vector dag is returned which specifies exac_y the same number of bits as in bytesize.
datatype - The datatype, e.g., pint, preal, pbit
parmpos - Parameter Position. This is the bit position of the start of this parm in the parm list. In the absence of bigvalparms the parmpos of the n'th formal parameter is simply 64*n. For example, parmpos = 64*1 for the first formal parm. When bigvalparms are present, parmpos must be adjusted to reflect the sizes of all previous bigvalparms.

Output:
The function value is (pexp.shl.pntrsiz) + a dag pointer for a vector lval. After the MULTI-WORD function value is computed, this lval dag pointer can be placed into dop1 of a vector opequal in order to effect the transfer of the multi-word function value to the location specified by the caller in the fake parameter. Normally the ple for this opequal dag is chained immediately before the ple created by rtn.

If datatype .eq. pbit, then a bit vector dag is returned.
Otherwise, a word vector dag is returned.

Operation:
The caller of a function which returns a multi-word value, adds one extra parameter to the parameter list usually at the beginning or the end. I suggest at the beginning for languages like C which must also handle variable length parm lists (the length of which are not known at compile time). Otherwise, at the end seems more reasonable (less confusing debugging at the machine language level). Strictly speaking, ffmlparm can be called before or after any of the calls to fmlparm. However, a reasonable implementation would only call it BEFORE all calls to fmlparm, or AFTER all calls to fmlparm.

This parameter contains the address of where the callee should store the multi-word function return value.

Ffmlparm creates a symbol table entry for this extra parameter. It then builds a vector lval referring to the extra parameter. This lval is the only access mechanism required by the front end in order to put the multi-word return value into the proper location.

This routine also chains the symbol table entry for the extra parameter under the opentry dag created by procinit.

Notes:
fvr%addr stands for Function Value Return ADDRess.

Ffmlparm creates a symbol table entry for this extra parameter.

FILE2REG

Addr of a word containing the FILE name to REGister.

Calling sequence:
lastple=file2reg(previous, fname)

Input:
previous - Index to the ple after which the texp ple's for this operation
should be chained.
fname - an array containing 8 ascii chars left-justified-space-filled
for the name of the current file.

Output:
The function value is the index to the ple for this operation.
Subsequent ple's should be chained after this one.

Operation:
Each machine instruction generated by the hybrid is the result of some source line in some file. Each time we start generating code due to a new file, file2reg may be called in order to store the addr of a word containing the file name into the file name register. This is strictly a debugging aid.

Note on internals: Fregdagp is a dag pointer to an
OPVAL-OPADDR-SYMBOL TABLE ENTRY subgraph for the file name register.
It already contains pexp.

FMLPARM

Formal parm. Add a parm to the formal parm list for this subprog.
Possible special action for bigvalparms.
Perform special action if the callee must make a copy of
a MULTI-WORD parm in its own AR.

Calling sequence:
stp1=fmlparm(name, numchars, datatype, formalmode, bytesize, parmpos)

Input:
name - an array containing ascii for the name of the formal parameter
which we are adding to the formal parameter list.
numchars - the number of non-blank characters in the name
datatype - stored into dt of the created symbol table entry.
formalmode - the mode of this formal
valparm - value formal parm
refparm - reference formal parm
callee_copy_parm - the parm list contains the address of a multi-word object and the CALLEE makes a copy of the object in the callee's AR.
bigvalparm - the caller puts a multi-word value right in the parm list. Note, when is_parent is nonzero, dummy symbol table entries will be made for each word in the big value parm (except the first).
bytesize - when formalmode=callee_copy_parm: bytesize is the size in bits of the parm. A full word vector transfer is generated. Thus the number of bits actually copied is equal to a multiple of 64.
- when formalmode=bigvalparm: bytesize is the size in bits of the parm. If bigvalparm is specified and common variable, is_parent, is nonzero, then we want to use the copiedarg feature of the backend to copy the entire multi_word value parm from the physical parm list into the local AR. However, civic won't do it for multi_word value parms without some scheduler changes (which are hard to come by these days). Thus, this routine builds enough dummy symbol table entries (with the copiedargb bit set) and chains them onto the opentry dag, so that the proper number of words from the physical parameter list are copied into the local AR. The dummy symbol table entries are of the form:
SourceParmName
SourceParmName<space>
SourceParmName<space><space>
etc.
This kludge is NOT considered wonderful.
o.w. - bytesize is undefined
parmpos - Parameter Position. This is the bit position of the start of this parm in the parm list. In the absence of bigvalparms the parmpos of the n'th formal parameter is simply 64*n. For example,

parmpos = 64*1 for the first formal parm. When bigvalparms are present, parmpos must be adjusted to reflect the sizes of all previous bigvalparms.
parmpos is only used if common variable, is_parent=0. See intrclic for details.
More notes: if this is a function and it returns a large function value, and in the language we are compiling, we choose to put the address of where to put the large function return value in the FIRST parameter position (as in C), then this must also be considered when computing parmpos.

Output:
name - when numchars is not a multiple of 8, name will be space filled out to a multiple of 8 chars.
The function value is the index to the (first) created symbol table entry for this formal parameter. When is_parent .ne. 0, then this index must be passed to getoff after the execution of backend because formal parameters will be allocated space (for the addr or the value) in the current AR so non-local references by nested procedures will be possible. Note, pvar is not included in the function value so it is not suitable for storing into the DOPn field of a dag.

Operation:
Fmlparm builds an appropriate symbol table entry for the formal parameter specified in name and chains it under the opentry dag created by procinit. Opcomma dags are created and chained as necessary. Because we are building the formal parm list top down, stoudag must be used. This is ok because detecting common subexpressions in formal parameter lists is not important. Fmlparm also increments the parm count in the opentry dag.

When formalmode=callee_copy_parm, then special action is required. Explicit dags and a ple are built to make the copy in the local AR.

When formalmode=bigvalparm, and common variable, is_parent is nonzero, then dummy symbol table entries must be generated. (See above).

Fmlparm is initialized by procinit. It should be called once for each formal parameter to the subprogram.

parmpos = 64*1 for the first formal

GETOFF

Get offset (for a local variable only, includes all formal parms)

Calling sequence:
call getoff(stp, symname, numchars, bitoffset)

Input:
stp - symbol table pointer (for a local variable only)

Output:
symname - An array containing the name of the local variable specified by stp.
numchars - the number of characters in this name.
symname and numchars are not used by real front ends. They are only present for debugging purposes and eventually should be deleted.

bitoffset - the bit offset into the local AR which was assigned to this variable by the Civic backend.

Operation:
One of the functions of the subroutine, backend, is to compute the offsets into the appropriate AR for all local variables in the current program or subprogram. These offsets must be remembered in case code in an inner scope references any of these variables as non-locals.

Civic symbol table entries for local variables are created by calls to functions, fmlparm and localvar. Getoff is called after backend to obtain the bit offset which was assigned by backend for each local variable. For a given local variable, the value supplied in parm, stp, is the same as the value returned earlier by fmlparm or localvar. The intended sequence of calls is:

Call fmlparm for each formal parm and remember the function values.
Call localvar for each local variable and remember the function values.
Call backend to compute offsets (and generate code of course).
Call getoff for all formal parms and local variables in order to obtain the offsets assigned by backend.

Getoff need not be called for procedures that do not contain nested procedures or spaces. This is because no one can reference the locals (including formals) as nonlocals.

An opinlfunc dag does not indicate an in-line function. An opinlfunc dag can

GETSTP

Get symbol table pointer.

Calling sequence:
stp1=getstp(name, numchars)

Input:
name - an array containing ascii for the name of the variable whose stp (civic symbol table pointer) we wish to obtain.

Output:
The function value is the symbol table pointer (without pvar) for the specified variable.

Operation:
It is assumed that the symbol table entry for the specified variable has already been created (for example, by localvar, comnvar, namecomn, makext, etc.)
Thus this routine thinks that it is merely finding the stp of an existing symbol table entry. No fields are set. If a symbol table entry for the given name does not already exist, this routine will build one and return the symbol table pointer. However, it will never know that this happened. It would be a preferable design if stosymn (or another parallel routine) indicated whether the symbol table entry was found or created. Then I could do more checking.

GETSTP

GLINIT

Global initialization
This routine is called exactly once to initialize the interface to the Civic backend.

Calling sequence:

call glinit(addr, srcfile, binfile, listfile, listopts, object,
optimizations, trace, arithmetic, statistics,
vectorization, vtmanagement, sensitive_ext, language,
ema_parm)

Input:

addr - The absolute address where the Civic backend can start its main data area. It is assumed that the Model heap has already been blocked from expanding and that the last addressable word in the controllee is at least addr + 3.

srcfile - 8 char ascii source file name, left justified space or null filled

binfile - 8 char ascii binary file name, left justified space or null filled

listfile - 8 char ascii listing file name, left justified space or null fill
SPECIAL CASE: If listfile .eq. "none " or
listfile .eq. "0" then
create no listing file. See civic file setup
near l command line processing

listopts - listing options, an ascii character string, left justified space or null filled, containing any of the Civic listing options (lo=) s, m, d, or x. Note: the options l and * are ignored. If none of these options is specified, the Civic default (corresponds to m) is used.

object - 0 means no object listing
1 means object listing, 1 column/page
2 means object listing, 2 columns/page

optimizations - Any mixture of ascii chars b,c,d,e,f,g,i
left justified space or null filled.
Each letter specifies a particular optimization.
If only one letter is specified, then it and all lower optimizations are selected.

trace - debug trace flags. Use exactly the same bit settings as the trc option on the Civic command line.

arithmetic - any mixture of ascii characters p, r, t, u, s, l
Same meaning as civic.

statistics - zero - no special action in intrterm
nz - intrterm will send compiler statistics to the screen.

vectorization - 0 or any mixture of the ascii chars t,a
left justified space or null filled.
same meaning as civic, i.e.:
t - Generate vector temporaries if necessary to vectorize

a loop.
a - "no dynamic equivalencing (in loops)".
From the civic help package: "Loops containing references to arguments or pointer[ed] variables will be vectorized".
(More precisely, the presence of arguments, or variables referenced through indirection will not necessarily block the vectorization of a loop).
Some additional notes on v=a.
The backend detects references to "variables via pointers" by observing indirection in the dags so the definition of v=a above is meaningful for languages other than lrltran.
The backend also checks to see if the "pointer base" for two pointered variables involved in an assignment are the same., e.g., x(i)=x(j). If they are the same, vectorization will still not occur unless option assert (nohazard) applies to the loop. Furthermore, if the compiler can detect that option assert (nohazard) is wrong, it will not vectorize anyway. The compiler can sometimes detect this when constants are involved in the subscript expressions.

vtmanagement - vector temporary management
the single ascii char i or s, LEFT JUSTIFIED, SPACE FILLED.
"i" - the variable iq8qsdp contains the bit address of the storage to be used for vector temporaries.
"s" - call the system memory manager to get space for vector temporaries.
If anything else is specified, it is treated as an error.

sensitive_ext - 0 - Externals are not case sensitive. External names will be mapped to UPPER CASE. The binaries will be flagged as case insensitive.
1 - Externals are case sensitive. External names will be in full upper-lower case. The binaries will be flagged as case sensitive.

language - "c " - the front end is for the C language.
"model " - the front end is for the model language.
There are codes in the UNICOS binary tables to specify "common" languages (like C) and "other" languages (like model). As of 8-26-86 that's all this is used for.

ema_parm - extended memory addressing
if code is being compiled for the Cray X-MP/48, special sequences are necessary to access data which is loaded at addresses greater than 2M words. Since that code carries a cost, the ema option allows control of when it is generated.

"0 " generate code assuming the field length of the running code will never exceed 2M words.

"d " generate code assuming common blocks may be located or extend

beyond address 2M, but all code blocks will be contained in the first 2M words.

"a " generate code so than any code or data may be loaded at addresses greater than 2M - temporarily this is implemented the same as "d "

Output:

srcfile, binfile, listfile - If these are null filled, this routine changes the nulls to spaces.

Side effects due to global initialization of the interface to the Civic backend.

Operation:

This routine does all the global initialization which must be performed only once before the Model front end calls the Civic backend for the first time.

HIGHUSE

High Usage - Flag this variable as a high usage variable, which if possible should be handled in a more efficient manner than normal.

Calling sequence:
call highuse(name, numchars)

Input:
name - an array containing ascii for the name of the variable (left justified zero or space filled).
numchars - the number of non-blank chars in the name.

Output:
name - when numchars is not a multiple of 8, name will be space filled out to a multiple of 8 chars.

Operation:
It is assumed that a symbol table entry for this variable has already been created via localvar. The ref field (reference count field) in the already existing symbol table entry is set to a relatively large value. It is expected that at some time in the future, the civic backend will handle variables with large reference counts in an especially efficient way, perhaps try to keep them in registers as long as possible. The initial motivation for this routine at the present time is to flag C "register" variables for special treatment. Note that "register" in C means "handle as efficiently as possible". It does NOT mean maintain this variable in a register.

Note:
This routine must be called AFTER localvar is called for the given name. Otherwise, the datatype field could be wrong. The interface can't check for this because unfortunately, the civic interfaces do not distinguish between finding an existing symbol table entry and creating a new one.

INLFNAME

Inline function name.
Make a symbol table entry for the name of an inline function; something suitable for dop1 of an opfcall or opinlfunc dag to DIRECTLY point at.

Calling sequence:
pvarstp1=inlfname(name, numchars)

Input:
name - an array containing ascii for the name of the inline function.
numchars - the number of non-blank chars in the name.

Output:
name - when numchars is not a multiple of 8, name will be space filled out to a multiple of 8 chars.
The function value is (pvar.shl.pntrsiz) + pointer to the found or created symbol table entry. This value is suitable for storing into the DOPn field of a dag.

Operation:
This routine finds or creates a symbol table entry for the given name and makes field settings appropriate for the name of an lrltran inline function.

INTRNSC

Intrinsic (See if this is the name of an lrltran intrinsic function that we can handle with a single opinlfunc or opfcall).

Calling sequence:
datatype=intrnsc(name, numchars)

Input:
name - one word containing the source language name of an lrltran intrinsic function (up to 8 chars, left-justified-space- or zero-filled).
numchars - the number of nonblank chars in the name

Output:
name - When numchars is not 8, name will be space filled out to 8 chars.
The function value is:
zero - if this is not an lrltran function that can be expressed in dags as a single opinlfunc or opfcall.

nonzero - if this is an lrltran function that can be expressed in dags as a single opinlfunc or opfcall. In this case the value is:
pint - if the function returns an integer.
preal - if the function returns a real.

Operation:
The interface has a partial table of lrltran intrinsic functions that can be expressed in dags by a single opinlfunc or opfcall. This table is searched and if a match is found for name, then the function return type is returned as the function value. If no match is found, then this function returns zero.

INTRTERM

Cleanup and termination of the hybrid

Calling sequence:
call intrterm(xx)

Input:
xx - same values as routine execute in file rest

Output:
none

Operation:
Final cleanup and termination.

KILVARS

Kill variables (Mark as dead all the scalar variables queued by qvar4kil)

Calling sequence:
lastple=kilvars(previous)

Input:
previous - a pointer to the previous ple.

Output:
The function value is a pointer to the newly created ple.

Operation:
This routine builds and chains a ple which marks all the variables queued by qvar4kil as dead variables. It then sets nextdeadvar to 1 which indicates that now there are no variables queued for kill.

LINE2REG

Line Number to Register. Line number to line number register.

Calling sequence:
lastple=line2reg(previous, linenum)

Input:
previous - Index to the ple after which the texp ple for this operation should be chained.
linenum - value of the current linenumber.

Output:
The function value is the index to the ple for this operation.
Subsequent ple's should be chained to this one.

Operation:
Each machine instruction generated by the compiler is the result of some source line in some file. Each time we start generating code due to a new source line, line2reg may be called in order to store this line number into the line number register. This is strictly a debugging aid.

Note on internals: Lregdagp is a dag pointer to an OPVAL-OPADDR-SYMBOL TABLE ENTRY subgraph for the line number register. It already contains pexp.

LITVAL

Literal val (Create an opval dag for a literal)

Calling sequence:
- dagp1=litval(binval, datatype)

Input:
binval - the binary value for the literal table entry to be created.
datatype - the datatype of the literal, e.g., poct, pint, preal.

Output:
The function value is (pexp.shl.pntrsiz) + a dag pointer to an opval dag pointing to a literal table entry for the value supplied in binval.

Operation:
The dag described above is found or created. Also the literal table entry to which the dag points is found or created.

LOCALVAR

Local variable
Make a symbol table entry for a local variable

Calling sequence:
stp1=localvar(name, numchars, datatype, elements, bytesize, static)

Input:
name - an array containing ascii for the name of the local variable
for which we are building a symbol table entry.
numchars - the number of non-blank chars in the name.
datatype - stored into dt of the found or created symbol table entry
elements - 0 if this object is NOT an array.
The total number elements if the object IS an array.
bytesize - bytesize (when datatype = pbyte)
When DATATYPE is:
pbyte - bytesize is the size in bits of the object being
declared. (For an array, bytesize is the size in
bits of each element in the array).
o.w. - bytesize is undefined
static - zero: This is NOT a static variable.
nz : This IS a static variable (save variable in civic parlance).

Output:
name - when numchars is not a multiple of 8, name will be space
filled out to a multiple of 8 chars.
The function value is the index to the found or created symbol table
entry. Note, pvar is not included in this value so it is not suitable
for storing into the DOPn field of a dag.

Operation:
This routine finds or creates a symbol table entry for the given name
and sets the defb field to force storage allocation and to preclude
a warning message that this variable has not been assigned a value.
This routine is intended to be used shortly after calling procinit
in order to build symbol table entries for all the local variables
to insure that they will all be assigned storage in case some inner
procedure references them.

Note:
localvar can be called a second time for a given name if you don't know
the values for elements and/or bytesize on the first call. (In that case
just use 0). All the other parms should be the same on both calls.
Two calls for the same name may be necessary due to C initializers, for
example, where we don't have the size information at the time we must build
the symbol table entry.

LVAL

Left Value. Left value for an assign statement (opequal) only.

Calling sequence:

dagp1=lval(name, numchars, level, bitoffset, varmode, offdagp)

Input:

name - an array containing ascii for the name of the variable for which we are building a left subgraph.

numchars - the number of non-blank chars in the name.

level - undefined for common variables. (Code as zero for efficiency).
- static nesting level of the declaration of the variable for which we are building a left subgraph.

bitoffset - for LOCAL AND COMMON VARIABLES: undefined
- for NON-LOCAL VARIABLES: bit offset within appropriate AR for start of the non-local object. This value was returned by getoff during the compilation of a previous subprogram (or the main program).

varmode - What kind of variable is this?
- notparm - non-parm
- valparm - value formal parm
- refparm - reference formal parm
- callee_copy_parm - the CALLEE made a copy of the parm in its own AR. Only used for certain multi-word objects.
- bigvalparm - a multi-word value formal (the whole value is right in the parm list).
- common_var - common variable

offdagp - (offset dag pointer):
- zero - no offset required
- nonzero - a dag pointer representing a WORD offset into the object specified by the prior parameters. This allows us to "index" into records, arrays, arrays of records, ect..

Output:

name - when numchars is not a multiple of 8, name will be space filled out to a multiple of 8 chars.

The function value is (pexp.shl.pntrsiz) + dag pointer to a dag subgraph which can be used for the lhs of an assigment statement only. Other contexts which require a left value must call address (or its successors if more are created after I write this). Several different subgraphs are produced depending on the values of level, varmode and offdagp.

Operation:
The subgraphs produced by lval are frequently occuring graphs. This routine returns a dag pointer to the appropriate kind of subgraph for the object named in input parm, name. When offdagp is nonzero, then we are referring to a word within the object.

In lval there are 6 basic types of scalar variable references. So in general, six different dag subgraphs are required to reference them. However, the dag subgraphs required for non-parm variables and value (formal) parms are identical (one for locals, another for non-locals) Hence, internally, lval generates only 4 different basic subgraphs. They are numbered 1 - 4 in the table below.
The dags become more complicated when offdagp is nonzero.

varmode	level = clevel local	level < clevel non-local
1 - non-parm	1	3
2 - value formal	1	3
3 - reference formal	2	4
4 - called makes copy of multi-word formal	1	3
5 - common variable		

Operation:

LVALBV

Left Value (bit vector). Left value for a bit vector.

Calling sequence:

```
dagp1=lvalbv(name, numchars, level, bitoffset,      varmode,
             bitoffdagp, lengthdagp)
```

Input:

```
name -----:
numchars -:
level ----:
bitoffset-: -- All the same as lval.
          :
varmode --:

bitoffdagp - (bit offset dag pointer)
             zero - no bit offset required
             nonzero - a dag pointer representing a bit offset into the
             object specified by the prior parameters. This allows us
             to "index" into records, arrays, arrays of records, etc.
lengthdagp - (pexp.shl.pntrsiz) + a dag pointer representing the
             length in bits of the bit vector.
             Note: As of sor130k, if the vector length evaluates to .le. 0,
             then no store is performed at run-time.
```

Output:

```
name - when numchars is not a multiple of 8, name will be space
       filled out to a multiple of 8 chars.
The function value is the same as lval except it refers to a bit vector.
This result dag may only be used where a vector
dag is legal (e.g., under an opequal with dshape=pvector).
```

Operation:

Parallel to lval. Note that bit vectors can start at arbitrary bit addresses. The addresses need not be multiples of 64.

LVALI

Left Value (indirect). Used ONLY on LHS of assignment.

Calling sequence:
dagp1=lvali(dagptr1, offdagp)

Input:
dagptr1 - (pexp.shl.pntrsiz) + a dag pointer representing the VALUE of a POINTER expression (this value is actually the address of the object pointed to).
offdagp - 0 or (pexp.shl.pntrsiz) + a dag pointer representing the word offset into the object pointed to.

Output:
The function value is (pexp.shl.pntrsiz) + a dag pointer representing the ADDRESS of the WORD at:
base address of object pointed to + word offset (from offdagp).
Note: This dag can only be used in dop1 of an assignment (opequal) dag. If you need such an address for other purposes, e.g., to put into a parm list, use addri.

Operation:
This routine provides a conceptually clear way to dereference a pointer expression.

Suppose you have a dag for the VALUE of a pointer expression, but what you really want is the address of some word relative to the beginning of the object pointed to. Lvali creates a dag which represents the desired address. This is a common operation when dereferencing pointer variables.

Special Case Operation:
Sometimes it is most natural for a front end to create an opadd dag representing the sum of the base address of an object and an index expression BEFORE it calls lvali. Such an opadd will block vectorization in the civic backend whereas an oplval representing the same sum will not block vectorization (the civic vectorizer explicitly looks for such oplvals). Thus if this routine detects that offdagp is 0 and dagptr1 refers to an opadd dag, then it will return an oplval dag that points to the same to operands as the opadd dag. By this means we hope to permit vectorization in cases where it might otherwise not occur. There is some question in my mind about whether this is the right place to reconstruct dags. Right now I'm really not sure and this is a rather safe and simple place to do it.

LVALIBV

Left Value Indirect (bit vector). Bit vector version of lvali, qv.

Calling sequence:
dagp1=lvalibv(dagptr1, bitoffdagp, lengthdagp)

Input:
dagptr1 - same as lvali
bitoffdagp - 0 or (pexp.shl.pntrsiz) + a dag pointer representing a bit offset into the object pointed to.

lengthdagp - (pexp.shl.pntrsiz) + a dag pointer for the length in bits of the bit vector.
Note: As of sor130k, if the vector length evaluates to .le. 0, then no store is performed at run-time.

Output:
The function value is the same as lvali except that it refers to a bit vector. This result dag may only be used where a vector dag is legal (e.g., under an opequal with dshape=pvector).

Operation:
Parallel to lvali. Note that bit vectors start at arbitrary bit addresses. These addresses need not be multiples of 64.

LVALIVEC

Left Value Indirect (vector). Multi-word version of lval, qv. (Implemented as a civic vector).

Calling sequence:
dagp1=lvalivec(dagptr1, offdagp, lengthdagp)

Input:
dagptr1 - same as lvali
offdagp - same as lvali

lengthdagp - (pexp.shl.pntrsiz) + a dag pointer for the length in words of the MULTI-WORD object.

Output:
The function value is the same as lvali except that it refers to a multi-word (civic vector) object. This result dag may only be used where a vector dag is legal (e.g., under an opequal with dshape=pvector).

Operation:
Parallel to lvali.

LVALT

Left Value Temporary. Left Value of a word in the hybrid temporary area. For LHS of assignment only.

Calling sequence:
dagp1=lvalt(offdagp)

Input:
offdagp - 0 or (pexp.shl.pntrsiz) + a dag pointer representing a word offset expression relative to the start of the hybrid temporary storage area.

Output:
The function value is (pexp.shl.pntrsiz) + a dag pointer representing the left value of the word in the hybrid temporary storage area at the offset represented by offdagp. This dag may ONLY be used on the LHS of an assignment statement.

Operation:
The result dag is computed based on the input parms.

LVALTBV

Left Value Temporary (bit vector). Left value of a bit vector.

Calling sequence:
dagp1=lvaltbv(bitoffdagp, lengthdagp)

Input:
bitoffdagp - 0 or (pexp.shl.pntrsiz) + a dag pointer representing a bit offset expression relative to the start of the hybrid temporary area.

lengthdagp - (pexp.shl.pntrsiz) + a dag pointer for the length in bits of the bit vector.
Note: As of sor130k, if the vector length evaluates to .le. 0, then no store occurs at run-time.

Output:
The function value is the same as lvalt except that it refers to a bit vector. This result dag may only be used where a vector dag is legal (e.g., under an opequal with dshape=pvector).

Operation:
Parallel to lvalt. Note that bit vectors start at arbitrary bit addresses. These addresses need not be multiples of 64.

LVALTVEC

Left Value Temporary (vector). Left value of a MULTI-WORD object, to be used only on LHS of assignment statement. (Implemented as civic vector).

Calling sequence:
dagp1=lvaltvec(offdagp, lengthdagp)

Input:
offdagp - same as lvalt

lengthdagp - (pexp.shl.pntrsiz) + a dag pointer for the length in words of the MULTI-WORD object.

Output:
The function value is the same as lvalt except that it refers to a multi-word (civic vector) object. This result dag may only be used where a vector dag is legal (e.g., under an opequal with dshape=pvector).

Operation:
Parallel to lvalt.

LVALVEC

Left Value (vector). Left value for a MULTI-WORD object, to be used only on the LHS of an assignment statement. (Implemented as a civic vector).

Calling sequence:

dagp1=lvalvec(name, numchars, level, bitoffset, varmode, offdagp, lengthdagp)

Input:

name -----:
numchars -:
level ----:
bitoffset-: -- All the same as lval.
:
varmode --:
offdagp --:

lengthdagp - (pexp.shl.pntrsiz) + a dag pointer for the length in words of the MULTI-WORD object

Output:

name - when numchars is not a multiple of 8, name will be space filled out to a multiple of 8 chars.
The function value is the same as lval except it refers to a multi-word (civic vector) object. This result dag may only be used where a vector dag is legal (e.g., under on opequal with dshape=pvector).

Operation:

Parallel to lval.