The DCode Intermediate Program Representation
Reference Manual and Report

John Gough[1]

Revision 2.1, December 5, 1994

[1]The original version of this report was extracted from the report on the compiler frontend **gp2d** (see reference 1). This report is maintained in electronic form on ftp server `pluto.fit.qut.edu.au` (Internet 131.181.2.16)

Abstract

This document describes the intermediate program representation *DCode*. The representation is an *ASCII* format which specifies instructions for an abstract stack machine. Interpretive code generators may process this file to produce object code for various target machines.

The semantics of each instruction in the code is explained, and the various directives detailed.

This document is intended to serve as a guide for those wishing to create new code generator backends for the existing compilers, and for those wishing to create frontends for a new language for the existing code generators.

The first part of the report is a concise reference manual for the language, while the second part is a detailed description of the way in which the language is used in typical cases. The second part describes the way in which the form is used in typical cases, and contains a number of examples.

Copyright

Revision History

Version 1.0, March 1993

The original document was extracted from the report on the portable Modula-2 frontend **gp2d**. Some additional information was added regarding the hints required by the existing backends.

The remaining references to **gp2d** in this document may be taken as examples of typical practice by conforming frontends rather than as a statement of how things *must* be done.

Version 1.1, April 1993

An additional variant of the *MKPAR* directive has been added, to allow compatibility with the calling conventions of Sun Microsystems C compilers. Minor corrections have been made also.

Version 1.2, June 1993

The features added for *SUN SPARC* compatability have been revised so as to fit with an improved method of handling the parameter abstraction. The separation between directives and instructions is now clearer. Primitives for supporting SSA have also been included.

Version 2.0 June 1994

Added support for 64-bit machines by making frontends responsible for the mapping from the logical types of the source language to the absolute sized types of the *DCode*.

The new *.EXPAND* is added for caller-copied open arrays. The #include *file* facility has been reintroduced, and the ability to perform one level of file inclusion in the *DCode* file for languages such as *C* which cannot specify the names of all imported symbols until the whole file has been read.

A new object declaration category *.DATA* has been introduced for initialized variable data. This allows the *.CONST* tag to be reserved for truly constant data.

Optional size indications have been added for imported and exported symbols, and also at the head of *.DATA* declarations. This has been necessary to allow *C* compilers on *MIPS* to interface with other *C* compilers which place data in different segements according to size.

Version 2.1 November 1994

A pragma format have been defined, and optional type-information strings added to local declarations for `gdb-stab` support.

Format of initialised data declarations is generalised to allow embedded labels. This facility is useful for declaring runtime type descriptors in object oriented languages.

The format used for supporting per-procedure exception handling has been revised to mark the exception handler entry point, and the retry label explicitly.

Contents

Chapter 1

Reference Manual

1.1 Overview

DCode is a stack based language for an abstract stack machine — the *D-Machine*. This machine is similar, in general terms to the well known P-Code and U-Code abstract machines. The main differences are due to the lower level at which address expressions are treated.

DCode has three components: interface information, data declarations, and procedure headers and code. The first part declares imports and exports of global symbols. The data declarations define the statically allocated data. Then for each procedure there is a header, with local declarations, and the code of that procedure's body. The instructions act on an abstract stack, pushing new values, popping values, and operating on the value(s) on the top of the stack. The abstract machine is a so-called "zero address machine". The only instructions which take memory addresses as arguments are those which push addresses onto the abstract stack, and those that call procedures at particular symbolic locations.

Values on the abstract stack are typed, but have only three types: *words, floats* and *doubles*. Objects of any primitive machine type may be pushed or popped. Operations such as `add` exist in different forms for each of these types. In the case of whole number arithmetic, and conversions between numeric formats, the overflow trapping mode, if any, is explicitly specified.

The *DCode* form described in this report is conventionally read and written to an ordinary *ASCII* file, and is designed to be readable by humans, as well as by language processors.

1.2 Lexical issues

In the *ASCII* representation of *DCode* line breaks are significant, and mark the end of various structures. Comments, and other white space has no syntactic significance, and may appear between any tokens. The other elements of the lexis are the keywords, identifiers, numbers and various other markers. Comments start with a semicolon, and end at the end of a line. As in Modula, character case is always significant. The lexical categories are defined in the table 1.1

alpha	=	{'a' .. 'z', 'A' .. 'Z', '$', '_'}
term1	=	{'"', *eoln*}
term2	=	{''', *eoln*}
digit	=	{'0' .. '9'}
octDigit	=	{'0' .. '7'}
hexDigit	=	*digit* ∪ {'A' .. 'F' }
alphanum	=	*alpha* ∪ *digit*
ident	→	*alpha alphanum** .
litstring	→	'"' *any* − *term1* '"' | ''' *any* − *term2* ''' .
number	→	['−' | '+'] *digit*+ | ['−' | '+'] *octDigit*+ 'B' | ['−' | '+'] *digit hexDigit** 'H' .
relop	→	'<' | '<=' | '>' | '>=' | '=' | '<>' | '#' .

Table 1.1: Character sets, and regular expressions for lexical categories.

By convention, comments which start with the exclamation character `!` are treated by code generators as pragmas. On backends which support this feature, the remainder of the line is copied to the output file in a backend-defined manner.

Identifiers obey the Modula conventions, except that dollar signs are allowed as valid characters, and lowline (underscore) characters may be freely used. By convention, frontends will prepend an underscore to all user-defined identifiers from the source code. Internally generated identifier names which do not have an underscore can therefore not collide with any user-defined external name. For systems in which the linker conventions do not use the prepended underscore the backend must strip this character from identifiers, before emitting the identifiers to the assembler.

Labels are lexically the same as identifiers, and defining occurrences are followed by a colon character. By convention, loop header labels are marked with the directive *.LOOP* in front of the label. Similarly, the ends of loops are marked with an *.ENDLOOP* directive. Simple backends may rely on this convention, see section 1.6.

Literal strings have the same format as in Modula, and cannot extend beyond a line end. They are enclosed in matching quote characters, which may be either single quotes ''' or double quotes '"'.

Some instructions have *relop* or *mode* arguments. The relops are "`<, <=, >=, >, =, <>, #`" as in Modula. For whole number arithmetic operations, and for numeric conversions, the mode may be *noTrap, crdOver, intOver* designating no overflow trap, unsigned, or signed trapping respectively. When the mode argument is optional, the default is always *noTrap.* In the case of division and remainder operations, the trapping mode is mandatory, as the signedness of the operators must be known in order to define the semantics of the operation.

Numbers are optionally signed, and obey Modula-2 lexical conventions. Numbers must begin with a decimal digit after the optional sign. Hexadecimal numbers end with the character "H", while octal numbers end with a "B". Decimal numbers require no suffix.

Starting with version 2.0 it is possible to include files verbatim in the *DCode* file. The format is —

`#include` *litstring*

The hash character must be the first character in the line, and the filename must appear as a literal string, that is, in single or double quotes. Only one level of inclusion is permitted.

1.2.1 Reserved words and predeclared identifiers

The keywords of *DCode* all begin with a period '.' and use all upper case characters. The keywords are—

.ADRS	.ALIGN	.ASCII	.ASCIIZ	.ASSEMBLY	.BITS16
.BITS32	.BYTE	.CHECK	.CONST	.COPY	.DATA
.DISPLAY	.DOUBLE	.ENDCASE	.ENDLOOP	.ENDP	.ENTRY
.EXCEPT	.EXPAND	.EXPORT	.FILE	.IMPORT	.JUMPTAB
.LOCAL	.LOOP	.NODISPLAY	.NOCHECK	.OPENCOPY	.PROC
.RETCUT	.RETRY	.SIZE	.TITLE	.TRAP	.VAR
.WORD					
crdOver	fpParam	intOver	noTrap		

The identifiers in the last line in the table are predeclared identifiers. These are context sensitive marks rather than reserved words. However, because of the underscore convention for user-defined identifiers they should never clash with other identifiers.

1.3 File syntax and semantics

The format of the *dcf* file is described by the extended *bnf* in the table 1.2.

1.3.1 Primitive data types

DCode knows of nine different primitive data types. Some of these are absolutely sized, so that it is the responsibility of every frontend to map the logical types of its source language into the types supported by *DCode*.

The types are *unsigned byte*, *signed byte*, *unsigned 16-bit word*, *signed 16-bit word*, *unsigned 32-bit word*, *signed 32-bit word*, *word*, *float*, and *double*. The *word* type is the "natural" word size of the machine, while other fixed types may be shorter.

1.3.2 The title

The title and file declarations specify the name of the module and the source file name from which it is derived. The title declaration has no particular function, except for identification purposes for the human reader. In contrast, the *.FILE* declaration is assumed to be the name of the source file, and should be written to the object file by all backends, for the benefit of debugger tools.

<table>
<tr><td>file</td><td>→</td><td>Title Declarations eofSy.</td></tr>
<tr><td>Title</td><td>→</td><td>.TITLE *ident* eolnSy .FILE *litstring* eolnSy.</td></tr>
<tr><td>Declarations</td><td>→</td><td>{Exports | Imports} {DataOrProc}.</td></tr>
<tr><td>Exports</td><td>→</td><td>.EXPORT NameList eolnSy.</td></tr>
<tr><td>Imports</td><td>→</td><td>.IMPORT NameList eolnSy.</td></tr>
<tr><td>NameList</td><td>→</td><td>*ident* [':' *number*] {',' [eolnSy] *ident* [':' *number*]}.</td></tr>
<tr><td>DataOrProc</td><td>→</td><td>.CONST [*number*] eolnSy {Label {ConstDec}}</td></tr>
<tr><td></td><td>|</td><td>.DATA [*number*] eolnSy {Label {ConstDec}}</td></tr>
<tr><td></td><td>|</td><td>.VAR eolnSy {Label VarDec}</td></tr>
<tr><td></td><td>|</td><td>[.LOCAL] .PROC HeaderInfo .ENTRY eolnSy
{Statement} {JumpTable} .ENDP eolnSy.</td></tr>
<tr><td>VarDec</td><td>→</td><td>.BYTE *number* [.ENTRY *number*] eolnSy</td></tr>
<tr><td></td><td>|</td><td>.BITS16 *number* [.ENTRY *number*] eolnSy</td></tr>
<tr><td></td><td>|</td><td>.BITS32 *number* [.ENTRY *number*] eolnSy</td></tr>
<tr><td></td><td>|</td><td>.WORD *number* [.ENTRY *number*] eolnSy</td></tr>
<tr><td></td><td>|</td><td>.DOUBLE *number* [.ENTRY *number*] eolnSy.</td></tr>
<tr><td>ConstDec</td><td>→</td><td>.BYTE *number* {',' *number*} eolnSy</td></tr>
<tr><td></td><td>|</td><td>.BITS16 *number* {',' *number*} eolnSy</td></tr>
<tr><td></td><td>|</td><td>.BITS32 *number* {',' *number*} eolnSy</td></tr>
<tr><td></td><td>|</td><td>.WORD *number* {',' *number*} eolnSy</td></tr>
<tr><td></td><td>|</td><td>.ASCII *litstring* eolnSy</td></tr>
<tr><td></td><td>|</td><td>.ASCIIZ *litstring* eolnSy</td></tr>
<tr><td></td><td>|</td><td>.ADRS *ident* [*number*] eolnSy.</td></tr>
<tr><td>HeaderInfo</td><td>→</td><td>ident "(" Arg {',' Arg} ")"eolnSy {CopyInfo} {LocalInfo}.</td></tr>
<tr><td>Arg</td><td>→</td><td>.NODISPLAY | .DISPLAY : *number* | .NOCHECK | .CHECK</td></tr>
<tr><td></td><td>|</td><td>.SIZE = *number* | .ASSEMBLY = *number* | .RETCUT = *number*.</td></tr>
<tr><td>LocalInfo</td><td>→</td><td>.LOCAL *ident number* ',' *number*
"(" *number* ',' *number* ',' *number* ")" ["fpParam"] [*litstring*] eolnSy.</td></tr>
<tr><td>CopyInfo</td><td>→</td><td>.COPY *number* ',' *number* .SIZE *number* [Align] eolnSy</td></tr>
<tr><td></td><td>|</td><td>.EXPAND ['('*number*')'] *number* ',' *number* .SIZE *number* [Align] eolnSy</td></tr>
<tr><td></td><td>|</td><td>.OPENCOPY ['('*number*')'] *number* .SIZE *number* [Align] eolnSy</td></tr>
<tr><td>JumpTable</td><td>→</td><td>.JUMPTAB Label eolnSy {IdList eolnSy}.</td></tr>
<tr><td>IdList</td><td>→</td><td>*ident* {',' [eolnSy] *ident*}.</td></tr>
<tr><td>Statement</td><td>→</td><td>[LabelTag] Label [OpCode [OpArg]] eolnSy</td></tr>
<tr><td></td><td>|</td><td>OpCode [OpArg] eolnSy</td></tr>
<tr><td></td><td>|</td><td>Directive eolnSy.</td></tr>
<tr><td>LabelTag</td><td>→</td><td>.EXCEPT | .LOOP | .RETRY .</td></tr>
<tr><td>Directive</td><td>→</td><td>.TRAP *ident',' ident* {',' OpArg}</td></tr>
<tr><td></td><td>|</td><td>.ENDLOOP [*number*]</td></tr>
<tr><td></td><td>|</td><td>.ENDCASE.</td></tr>
<tr><td>Label</td><td>→</td><td>*ident* ':' .</td></tr>
<tr><td>OpArg</td><td>→</td><td>*number* {',' *number*} ["fpParam"]</td></tr>
<tr><td></td><td>|</td><td>*mode* | *relop*</td></tr>
<tr><td></td><td>|</td><td>*ident* [*number*].</td></tr>
<tr><td>Align</td><td>→</td><td>.ALIGN *number* .</td></tr>
</table>

Table 1.2: The extended bnf for the input file.

1.3.3 Declarations

Imports and exports

The declarations part of the file begins with imports and exports. Exported names are visible to the linker, while imported names are expected to be resolved to locations external to the module.

Declarations	→	{Exports | Imports} {DataOrProc}.
Exports	→	.EXPORT NameList eolnSy.
Imports	→	.IMPORT NameList eolnSy.
NameList	→	*ident* [':' *number*] {',' [eolnSy] *ident* [':' *number*]}.

The elements of name lists are comma separated, and may contain embedded end of lines. The optional colon and number give the storage size of the imported or exported object. A zero number has the default semantics. Knowing the size of imported objects is important for machines with conventions which place small objects in a separate data segment.

Data declarations

Variables and constant declarations appear after imports and exports, and may alternate freely with procedure declarations.

Data may be declared in three ways. Constant declarations declare values which, where possible, will be placed by backends in a readonly memory segment. Data declarations using the *.DATA* keyword are initialized variables, and will be placed in a read-write memory segment. Finally variable declarations request uninitialized space, conventionally in the *BSS* memory segment.

DataOrProc	→	.CONST [*number*] eolnSy {Label {ConstDec}}
	|	.DATA [*number*] eolnSy {Label {ConstDec}}
	|	.VAR eolnSy {Label VarDec}

Constant and initialized data declarations may be specified in any convenient way. Backends usually ensure that labels are word-aligned, so that the data may be specified byte-by-byte, or in larger units. There is no provision for handling floating point representations, so that frontends must determine the representation of floating point constants.

Note that a multiple labels may be interspersed within declarations of constant or initialized data blocks. The optional number, if present, gives the total size of the following block.

ConstDec	→	.BYTE *number* {',' *number*} eolnSy
	|	.BITS16 *number* {',' *number*} eolnSy
	|	.BITS32 *number* {',' *number*} eolnSy
	|	.WORD *number* {',' *number*} eolnSy
	|	.ASCII *litstring* eolnSy
	|	.ASCIIZ *litstring* eolnSy
	|	.ADRS *ident* [*number*] eolnSy.

The *.ASCII* declaration declares an initialized string, while the *.ASCIIZ* declaration appends a nul (zero) character to the literal string which is specified.

Variable declarations do not define the values of the bytes or words which are specified, but rather specify the *number* of units which are required. Note that a label is mandatory with each variable declarations, but multiple variables may be declared following the *.VAR* keyword.

VarDec → .BYTE *number* [.ENTRY *number*] eolnSy
| .BITS16 *number* [.ENTRY*number*] eolnSy
| .BITS32 *number* [.ENTRY*number*] eolnSy
| .WORD *number* [.ENTRY *number*] eolnSy
| .DOUBLE *number* [.ENTRY *number*] eolnSy.

Variable declarations guarantee that a contiguous region of memory of the specified size will be allocated in the one memory segment. However, backends need not guarantee that variables will be set out in memory in the order in which they are declared in the *DCode*. Thus, if it is required that the label be placed in the interior of a region of memory, then the optional form with an *.ENTRY* declaration must be used. The entry declarations specifies the distance in bytes from the start of the contiguous region at which the label will be placed.

1.4 Procedure declarations

Procedure declarations consist of the procedure header, local declarations, the procedure body, followed perhaps by jump tables for case or switch statements.

DataOrProc → ... | [.LOCAL] .PROC HeaderInfo .ENTRY eolnSy
{Statement} {JumpTable} .ENDP eolnSy.

Procedure declarations begin with the marker keyword *.PROC* and end with the keyword *.ENDP*. The header has the declared name of the procedure (which may or may not correspond to an exported name).

The *.PROC* keyword may be preceded by the optional keyword *.LOCAL*. If present, this marker states that the procedure is neither exported or assigned as a procedure variable. For some machine architectures faster calls may be made to such procedures.

1.4.1 Procedure headers

HeaderInfo → ident "(" Arg {',' Arg} ")" eolnSy {CopyInfo} {LocalInfo} .

Procedure headers declare various attributes of the procedure which have been evaluated by the frontend. There are three parts, the header proper, copy information for any value parameters passed by reference, and local variable information.

The header information

The arguments of the procedure header specify properties of the procedure prolog and epilog.

Arg → .NODISPLAY | .DISPLAY : *number* | .NOCHECK | .CHECK
| .SIZE = *number* | .ASSEMBLY = *number* | .RETCUT = *number*.

The conventional method used by *DCode* compilers to access uplevel data uses a display vector. The keywords *.DISPLAY, .NODISPLAY* indicate whether or not a display location needs to be updated. In the case that the display is needed the level is indicated by the notation—

`.DISPLAY` : *lexLevel*

Compilation unit bodies are at lexical level 0, and require no display. The default is no display, which in turn is the same as display at level zero. If a procedure has no objects which are accessed from nested scopes, then no display updating is required, and it is permissible to declare *.NODISPLAY.*

The keywords *.CHECK, .NOCHECK* in the procedure header indicate whether or not a stack overflow check is to be performed. Checking is the default, so the check keyword is usually included simply for clarity.

The *.SIZE = number* phrase declares the size of the (fixed part of the) stackframe. The number following the equality is the size in bytes. The default size is zero. The size includes any callee copied structures of fixed size, but does not include conformant arrays.

The *.ASSEMBLY = number* phrase declares the size of the parameter assembly area in the stack frame. This is not needed for architectures where parameters are pushed onto a runtime stack. The number following the equality is the size in bytes. The default size is zero.

The *.RETCUT = number* phrase declares that this procedure uses "Pascal call conventions". Such procedures are expected to remove *number* bytes from the runtime stack during the exit epilog.

Local variable information

Local variables are declared by specifying their identifier, stack frame offset, size, and other attributes. The first number following the identifier is the stack frame offset, while the second is the size of the variable in bytes.

LocalInfo → .LOCAL *ident number* ',' *number*
"(" *number* ',' *number* ',' *number* ")" ["fpParam"] [*litstring*] eolnSy.

The attribute information is a triple of numbers each of which may be only a single zero or one digit. In the first position a one specifies that the corresponding variable is accessed by a nested procedure. In the second position a one specifies that the corresponding variable has its value threatened[1] by a nested procedure. The final attribute in the parentheses is one to specify that the variable has its address taken within this procedure. Conventionally, structures and unions are given a one attribute in the final position to ensure that the backend will not attempt to place such a variable in a register.

The optional *fpParam* marker specifies that the variable is a floating point datum. This marker should be generated by any frontends which are intended to be portable, since some architectures will be unable to locate floating point parameters without this hint. It is permissible, and perhaps helpful to use this marker for non-parameter local variables.

[1] A variable is threatened by a procedure if the procedure directly assigns to the variable, or if takes the address of the variable or passes the address as a parameter to another procedure.

The optional literal string at the end of the declaration allows type information to be inserted so that the code generator can create debugger type information for the local variable.

Parameter copy information

CopyInfo	→	.COPY *number* ',' *number* .SIZE *number* [Align] eolnSy
	|	.EXPAND ['('*number*')'] *number* ',' *number* .SIZE *number* [Align] eolnSy
	|	.OPENCOPY ['('*number*')'] *number* .SIZE *number* [Align] eolnSy
Align	→	.ALIGN *number* .

Parameter copy information is required in two circumstances. Firstly, if value parameters are passed by reference and are copied by the called procedure then this must be specified. Secondly, if conformant array parameters are either copied by the called procedure or are assembled by the caller at runtime, then information must be given to allow the array to be copied or the correct amount of space allocated respectively.

The *.COPY* declaration specifies that a value parameter has been passed by reference, and must be copied by the called procedure. The arguments to the declaration are —

`.COPY` *paramOffset* , *destOffset* `.SIZE` *objectSize* `.ALIGN` *alignNum*

The first argument is the offset in the stack frame of the parameter which is the reference to the actual parameter. The second is the destination offset in the fixed part of the declared stack frame. The declared size is the number of bytes in the object to be copied.

The alignment specification may be omitted if the target machine has no alignment constraints, or if the object is byte aligned. Otherwise the alignment number would be 1, 2, 4 or 8 for contemporary machines.

The copying of value mode, variable size, open arrays parameters is indicated as follows —

`.OPENCOPY(`*dimensions*`)` *paramOffset* `.SIZE` *elementSize* `.ALIGN` *alignNum*

The meaning of the alignment attribute is as for fixed size array copies. However, in this case, the destination address must be determined at runtime. The size of the copy is determined by the computation —

$$size = elSize \times \prod_{i=1}^{dim} (high_i + 1)$$

where *elSize* is the specified element size, *dim* is the number of dimensions and $high_i$ is the i-th high value. The actual parameter reference is at stack frame offset *paramOffset* and it is assumed that the high value(s) are word-sized and at consecutive postions in the stack frame.

After the array has been copied, the backend must overwrite the reference so as to point to the copy. The allocated space need not be in the stack frame, but in any case the procedure epilog is responsible for deallocating the space at exit.

The *.EXPAND* declaration specifies that the procedure prolog will allocate and deallocate space for an open array of a shape specified by a designated actual parameter reference, the *template array*. The declaration is —

`.EXPAND(`*dimensions*`)` *paramOfst, dstPtrOfst* `.SIZE` *elemSize* `.ALIGN` *alignNum*

As for open array copies, the size of the allocated space is given by —

$$size = elSize \times \prod_{i=1}^{dim} (high_i + 1)$$

where *elSize* is the specified element size (which is not necessarily the same as the element size of the template array), *dim* is the number of dimensions and $high_i$ is the i-th high value. The template array reference is at stack frame offset *paramOffset* and it is assumed that the various *high* values are contiguous with that reference.

In this case, after the space has been allocated, the address of the allocated space will be placed in the stack frame at the offset given by *dstPtrOfst*. The allocation of an offset to hold this pointer is the responsibility of the frontend.

1.4.2 Procedure bodies

Procedure bodies begin with the keyword *.ENTRY* and end either with jump table declarations, or with the *.ENDP* marker. The form of the statements which make up the body of the procedures is treated in section 1.5.

1.4.3 Jump tables

Case and switch statements are implemented by a vectored jump through a jump table. Jump tables appear at the end of procedures, immediately preceding the *.ENDP* marker.

JumpTable	→	.JUMPTAB Label eolnSy {IdList eolnSy}.
IdList	→	*ident* {',' [eolnSy] *ident*}.

Jump tables consist of a label, by which the jump table is referenced, and an identifier list corresponding to all of the labels which the table references.

All of the identifiers in the list correspond to labels local to the current procedure body. The list is ordered so that consecutive identifiers in the list correspond to the labels which will be selected by consecutive values of the select expression index.

1.5 Statements

The body of procedures consists of statements and directives, one per line. Statements have optional labels, and labels may appear alone on lines.

Statements are the instructions of the *DCode* form, together with a small number of directives. The statements operate on the stack of the abstract machine, taking any non-literal operands from the stack, and returning results to the stack.

Statement	→	[LabelTag] Label [OpCode [OpArg]] eolnSy
	|	OpCode [OpArg] eolnSy
	|	Directive eolnSy.
LabelTag	→	.EXCEPT | .LOOP | .RETRY .
Directive	→	.TRAP *ident*',' *ident* {',' OpArg}
	|	.ENDLOOP [*number*]
	|	.ENDCASE.
OpArg	→	*number* {',' *number*} ["fpParam"]
	|	*ident* [*number*]
	|	*relop*
	|	*mode* .
Label	→	*ident* ':' .

DCode knows of nine different primitive data types, as detailed in section 1.3.1. However, the abstract stack itself knows only three: *words, floats* and *doubles*. For existing implementations, the first corresponds to either 32– or 64–bit words, and the other two to single and double precision *IEEE* floating point formats respectively. Whenever a datum is pushed on the abstract stack, the type of the source datum is specified and any format conversion, such as zero extension or sign extension, is implicit.

The semantics of statements are for the most part self explanatory, but some more detailed discussion is required for the division operators, testing and trapping, the handling of the parameter abstraction, and the handling of off-stack temporaries.

1.5.1 Directives

Marker directives are placed in the code to pass information to the backends. The directives are as follows —

`.ENDCASE` this directive marks the end of a case statement, and conventionally follows the end of the default (*ELSE*) case.

`.ENDLOOP` this directive marks the end of any loop. The number following the directive, if present, has a meaning which is defined by the frontend, for the benefit of the human reader. For example, it might indicate the loop label with which it pairs.

`.EXCEPT` This label tag declares that the tagged label is the entry point to which control is transfered if an exception is raised in the preceding procedure body.[2]

`.LOOP` this label tag marks the start of a loop. The directive is attached to the label to which the back-edge(s) of the loop jump(s).

`.RETRY` this label tag marks the point to which control is transferred following the execution of a retrial attempt in an exception handler.

`.TRAP` this directive is used to indicate to backends that an out-of-line trap call is required, at the end of the current procedure.

[2]This implies that exception handling must be on a per-procedure basis. More capable mechanisms are currently under consideration for later revisions of this intermediate form.

Semantics of the trap directive

The trap directive is a declaration, and may logically occur anywhere within the body of the procedure to which is refers. It has two mandatory arguments, both of which are identifiers, and as many as four optional arguments. The form allows backends to create low overhead, out-of-line trap calls to runtime system entry points which are unknown to the backend.

The mandatory arguments of the directive are the trap identifier, and the identifier associated with the out-of-line label. Following the mandatory arguments as many as four optional arguments may occur. These are of the form of identifiers (with optional numeric offsets), or numbers.

At runtime, when a branch is taken to the label specified by the second argument to a trap directive, control passes to an out-of-line sequence of machine instructions. In this sequence the specified, optional arguments (third, ... arguments to the directive) will be passed (as first, ... parameters to a call) to the runtime trap entry point specified by the first argument to the directive. It is assumed that control will never return from these traps.

1.5.2 Instructions

Arguments to the instructions

The majority of abstract stack machine instructions have no explicitly specified operands. Exceptions include the *push address* and *call procedure* instructions, these take a symbolic address as operand. In all such cases the symbolic address argument has as its most general form *ident* $\pm$ *number*.

Some other instructions, such as *push literal* and *add offset* take numeric arguments. As noted in section 1.2 such numeric tokens take an optional sign.

Most arithmetic instructions take an optional or mandatory mode argument. These arguments are one of *noTrap, intOver, crdOver* denoting no overflow trapping, integer overflow detection or cardinal (unsigned) overflow detection. The default is no trapping.

For the division instructions, the trapping mode is mandatory, and must be either *intOver* or *crdOver*, in order to completely specify the semantics of the operations.

Finally, the floating point comparison instructions have an argument which specifies the relational operator to be tested. The relops are "`<, <=, >=, >, =, <>, #`" as in Modula.

Division and remainder instructions

There are a wide variety of division and remainder instructions, sufficient to cater for all the common high-level languages. The definitions of these instructions is as follows —

$$\begin{aligned} a\,\text{`}\mathtt{div}\text{'}\,b &= \lfloor a/b \rfloor \\ a\,\text{`}\mathtt{mod}\text{'}\,b &= a - (a\,\text{`}\mathtt{div}\text{'}\,b) \times b \\ a\,\text{`}\mathtt{slash}\text{'}\,b &= \mathcal{R}_0(a/b) \\ a\,\text{`}\mathtt{rem}\text{'}\,b &= a - (a\,\text{`}\mathtt{slash}\text{'}\,b) \times b \end{aligned}$$

where the / operators on the right denote exact division, and where $\mathcal{R}_0 : \mathcal{R} \rightarrow \mathcal{I}$ is the *round toward zero* function.

Here is a table, which gives an example of the results of applying the various division and remainder operations to some typical operands on the stack. In each case the top of stack is the right operand for the operation, while the left operand is below that.

stack top	10	–10	10	–10
next elem.	31	31	–31	–31
`slash`	3	–3	–3	3
`rem`	1	1	–1	–1
`div`	3	–4	–4	3
`mod`	1	–9	9	–1

The `test` instruction

The test instruction allows frontends to encode such things as index and range tests without inline expansion in the intermediate code. This has some advantages with target architectures which possess test-and-trap instructions. In some cases, it is also advantageous to keep tests as indivisible primitives in order to prevent the unnecessary fragmentation of basic blocks. The arguments to the instruction are the name of the trap to call if the test fails, and the low and high bounds against which to test.

Handling the parameter abstraction

The passing of parameters to procedures is achieved by two instructions `mkPar` and `blkPar`.

`mkPar` *par-size, asm-offset* [`fpParam`]

The effect of the `mkPar` instruction is to pop the abstract stack and move the popped datum to the parameter location. Typically this location will be the top of the runtime stack, or some specified location in the parameter assembly area, or perhaps some machine register. The first argument to the instruction is the size of the datum,[3] while the second argument, *par-offset* is the offset in the parameter assembly area. In the case of backends which pass parameters on a runtime stack, it is permissible for the second argument to be an arbitrary number. The optional `fpParam` marker indicates that the datum is a floating point value.

`blkPar` *par-size, asm-offset, opt-alignment*

The effect of the `blkPar` instruction is to copy *par-size* bytes from the location pointed to by the top of the abstract stack, and to move this datum to the parameter location. The address is popped from the abstract stack. Typically the parameter location will be the top of the runtime stack, or some postion in the parameter assembly area starting at offset *par-offset*. In the case of backends which pass parameters on a runtime stack, it is permissible for the second argument to be an arbitrary number. An optional third argument specifies the source alignment.

[3]Of course, the size of the datum could have been computed by an interpretive code generator in the backend, but is specified explicitly for simplicity.

In any case, the details of which parts of the parameter assembly area (if any) are moved into registers is transparent to the *DCode* form. However, the method of parameter assembly cannot be hidden from the frontend, since this constrains the order of parameter evaluation.

Returning function values

Function values are communicated by means of the `popRet`X`, pshRet`X abstraction. The first of these pops the top of stack *word, float* or *double* to the return location. This is typically a machine register. The `pshRet`X instructions take the value in the return location and push it onto the abstract stack. In the case of short values, the exact sign or zero extension is specified.

DCode provides some choice in the way that structured function values are returned. The default is to require the frontend to allocate space for the return value, and to pass a reference to this location as an implicit, additional, first parameter to the function. Using this method does not require any special instructions. However, for machines for which it is conventional to pass such function return-value pointers in a dedicated location in the runtime stack frame an optional abstraction has been introduced. The `mkDstP` instruction pops the top of the abstract stack into the dedicated function return-value pointer location. Within the called procedure the function return-value pointer is accessed by means of the `pshDstP` instruction.

Off-stack temporaries

DCode provides for values to be moved from off the stack of the abstract machine into off-stack locations. There are two separate mechanisms for doing this.

The `mkTmp, pshTmp` instruction pair store and retrieve temporary values off the stack. The *make-temporary* instruction non-destructively copies the datum on the top of stack into an off-stack location. The instruction specifies the absolute value of the offset in the stack frame which the frontend has temporarily reserved for this datum. Corresponding *push-temporary* instructions will push the value onto the abstract stack. Several points need to be made about this mechanism. Code generating backends will attempt to store such temporaries in machine registers, and are not obliged to use the same register for each defining occurrence of a temporary at a particular offset. The mechanism should therefore not be used to attempt to merge values computed along different control paths in a program. Furthermore, interpretive code generators may treat certain temporaries as symbolic quantities, so that the temporary has no concrete existence at runtime, either at the stack frame location which was specified, or in a machine register.

The `popFi, pshFi` instruction pair store and retrieve values in off-stack locations which are determined by the backend. The *pop*-Φ instruction pops the top of the abstract stack into an off-stack location which is allocated by the backend. The argument to the instruction is simply an index which uniquely selects a particular location. Every such instruction with the same index within any single procedure, will move the top of stack to the same location. The *psh*-Φ instruction pushes the saved value back onto the abstract stack.

This instruction pair is suitable for merging values which are computed along different control paths in a program, as is required by language C's conditional expressions. However, it is preferable in all such cases for the frontend to allocate a dummy local variable, with '(0,0,0)' attributes (see section 1.4.1), and merge values by assigning to this variable. This attribute assignment will ensure that the variable is considered for allocation to a machine register.

Alphabetical instruction list

In the following table parameters in square brackets mean that the argument is optional. Instructions marked with a dagger sign † are instructions which have to do with the specifics of segmented architectures such as the iapx86, and might be ignored in other versions.

opcode	params	description
`abs`	[mode] ...	Takes the integer on the top of the stack and replaces it by its absolute value. If mode is *intOver* then an attempt to negate the minimum integer is trapped
`absDbl`	....	Takes the double precision floating point number on the top of the stack and replaces it by its absolute value
`absFlt`	....	Takes the single precision floating point number on the top of the stack and replaces it by its absolute value
`add`	[mode] ...	Adds the two values on the top of the stack with overflow trapping semantics as specified by mode. The stack is popped by two, and the result pushed
`addAdr`	....	Adds the value on the top of stack to the address below that. The addend may be either positive or negative, and the computation is performed modulo the size of the address space. The stack is popped by two, and the result pushed
`addDbl`	....	Adds the two *double*s on the top of stack. The stack is popped by two, and the result pushed
`addFlt`	....	Adds the two *float*s on the top of stack. The stack is popped by two, and the result pushed
`addOff`	offset	Adds the constant offset to the address on the top of the stack. The offset may be either positive or negative, and the computation is performed modulo the size of the address space. The new address value replaces the old on the top of stack
`andWrd`	....	Bitwise *AND* of two top of stack values. The stack is popped by two, and the result pushed
`assign16`	....	Assigns the least significant 16-bits of the value second on the stack to the address specified on the top of stack. The stack is popped by two
`assign32`	....	Assigns the least significant 32-bits of the value second on the stack to the address specified on the top of stack. The stack is popped by two
`assignB`	....	Assigns the least significant *byte* of the value second on the stack to the address specified on the top of stack. The stack is popped by two
`assignD`	....	Assigns the *double* second on the stack to the address specified on the top of stack. The stack is popped by two
`assignF`	....	Assigns the *float* second on the stack to the address specified on the top of stack. The stack is popped by two

opcode	params	description
`assignW`	....	Assigns the *word* second on the stack to the address specified on the top of stack. The stack is popped by two
`bitNeg`	....	Bitwise negate the top of stack value. The result replaces the top of stack value
`blkCp`	[number] .	Block copy. This instruction has three parameters on the stack: on the top of stack is the number of bytes of the block, below that is the source address, below that the destination address. All three values are popped. The optional number is 1, 2, 4 or 8 and gives the block alignment. The default alignment is 1 (byte aligned)
`blkPar`	number, number [,number]	This instruction is performs a parameter block copy. The top of the stack is a pointer to the actual parameter. The first two arguments to this directive are the parameter size, and the destination location in the parameter assembly area. An optional third parameter gives the *source* alignment boundary, and is always 1, 2, 4 or 8. In the case of machines which pass their parameters on the stack, the number of bytes specified will be copied to the runtime stack. In the case of fixed parameters the block will be copied to the specified offset from the present stack pointer. In either case, the stack is popped by one
`boolNeg`	....	Negates the Boolean value on the top of stack. The result replaces the top of stack value
`brFalse`	label	Branch if the top of stack value is zero, and pop the stack
`brTrue`	label	Branch if the top of stack value is non-zero, and pop the stack
`branch`	label	Branch unconditionally
`call`	identifier, number...	Call the procedure designated by the identifier, with the specified number of parameters
`crdGE`	....	Compares two top values as unsigned words. Returns *TRUE* if the second from the top is greater than or equal to the top. The stack is popped by two
`crdGT`	....	Compares two top values as unsigned words. Returns *TRUE* if the second from the top is greater than the top. The stack is popped by two
`crdLE`	....	Compares two top values as unsigned words. Returns *TRUE* if the second from the top is less than or equal to the top. The stack is popped by two
`crdLS`	....	Compares two top values as unsigned words. Returns *TRUE* if the second from the top is less than the top. The stack of the abstract machine is popped by two
`cutPars`	number...	Remove parameter bytes (only for machines which pass parameters by pushing them onto the stack of the actual machine – the *concrete stack*)
`dblRel`	relop	Pop and compare *double*s on top of stack, and push the Boolean result. Right operand is initially top of stack
`derefF`	....	Dereferences the pointer to *float* on the top of stack. The top of stack value is replaced by the result

opcode	params	description
`derefD`	....	Dereferences the pointer to *double* on the top of stack. The top of stack value is replaced by the result
`derefSB`	....	Dereferences the pointer to *signed byte* on the top of stack. The value on the top of the stack is replaced by the dereferenced value
`derefS16`	....	Dereferences the pointer to *signed 16-bit word* on the top of stack. The value on the top of the stack is replaced by the dereferenced value
`derefS32`	....	Dereferences the pointer to *signed 32-bit word* on the top of stack. The value on the top of the stack is replaced by the dereferenced value
`derefUB`	....	Dereferences the pointer to *unsigned byte* on the top of stack. The value on the top of the stack is replaced by the dereferenced value
`derefU16`	....	Dereferences the pointer to *unsigned 16-bit word* on the top of stack. The value on the top of the stack is replaced by the dereferenced value
`derefU32`	....	Dereferences the pointer to *unsigned 32-bit word* on the top of stack. The value on the top of the stack is replaced by the dereferenced value
`derefW`	....	Dereferences the pointer to *word* on the top of stack. The value on the top of the stack is replaced by the dereferenced value
`div`	mode.....	Performs the div operation on the two values on the top of the stack with semantics as specified by mode. The right operand is on top. The stack is popped by two, and the result pushed
`divDbl`	....	Divides the *double*s on the top of stack. The right operand is on top. The stack is popped by two, and the result pushed
`divFlt`	....	Divides the *float*s on the top of stack. The right operand is on top. The stack is popped by two, and the result pushed
`dRound`	[mode] ...	Convert *double* to word rounding to nearest. If mode is *intOver* or *crdOver* then values outside the (respectively) signed or unsigned range are trapped. If trapping is not specified, then backends should produce the correct value if possible. The top of stack value is replaced by the result
`dFloor`	[mode] ...	Convert *double* to word rounding to minus infinity. If mode is *intOver* or *crdOver* then values outside the (respectively) signed or unsigned range are trapped. If trapping is not specified, then backends should produce the correct value if possible. The top of stack value is replaced by the result
`dToFlt`	....	Shorten *double* on top of stack to *float*. The top of stack value is replaced by the result
`dTrunc`	[mode] ...	Convert *double* to word rounding to zero. If mode is *intOver* or *crdOver* then values outside the (respectively) signed or unsigned range are trapped. If trapping is not specified, then backends should produce the correct value if possible. The top of stack value is replaced by the result
`dup1`	....	Duplicates the top value on the stack
`endF`	....	Marks the end of the *DCode*s

opcode	params	description
endP	....	Marks the end of the current procedure
exit	....	Unconditional jump to the procedure epilog
†flatten	....	Transforms the segmented address on the top of stack into a unsigned (*nop* except for iapx86 in 16-bit segmented mode)
fltRel	relop	Pop and compare *float*s on top of stack, and push the Boolean result. Right operand is initially top of stack
fRound	[mode] ...	Convert *float* to word rounding to nearest. If mode is *intOver* or *crdOver* then values outside the (respectively) signed or unsigned range are trapped. If trapping is not specified, then backends should produce the correct value if possible. The top of stack value is replaced by the result
fFloor	[mode] ...	Convert *float* to word rounding to minus infinity. If mode is *intOver* or *crdOver* then values outside the (respectively) signed or unsigned range are trapped. If trapping is not specified, then backends should produce the correct value if possible. The top of stack value is replaced by the result
fToDbl	....	Widens a *float* to a *double*. The top of stack is replaced by the result
fTrunc	[mode] ...	Convert *float* to word rounding to zero. If mode is *intOver* or *crdOver* then values outside the (respectively) signed or unsigned range are trapped. If trapping is not specified, then backends should produce the correct value if possible. The top of stack value is replaced by the result
intGE	....	Compares two top values as signed words. Returns *TRUE* if the second from the top is greater than or equal to the top. The stack of the abstract machine is popped by two
intGT	....	Compares two top values as signed words. Returns *TRUE* if the second from the top is greater than the top. The stack of the abstract machine is popped by two
intLE	....	Compares two top values as signed words. Returns *TRUE* if the second from the top is less than or equal to the top. The stack of the abstract machine is popped by two
intLS	....	Compares two top values as signed words. Returns *TRUE* if the second from the top is less than the top. The stack of the abstract machine is popped by two
iToDbl	....	Floats the top of stack from integer to *double*. The top of stack is replaced by the result
iToFlt	....	Floats the top of stack from integer to *float*. The top of stack is replaced by the result
jump	....	Branch to the address on the top of the stack. Performs a case statement jump (more details in report). The stack is popped
lineNum	number...	Marks the line number for diagnostics only. Typically generated by compilers in response to the **–g** flag
†makeAdr	....	Transform the top of stack unsigned into an address, this is the inverse of `flatten`

opcode	params	description
`mkDstP`	number...	This instruction pops the value from the top of the stack, and moves it to the dedicated destination pointer location. This is only used on architectures such as *SPARC* which do not pass destination pointers as an additional ordinary parameter. The argument is the destination size, which is used in some conventions as an attribute in a trap instruction
`mkTmp`	number...	Associates the top of stack value with the current stack frame location \$fp − *number*. In most backends symbolic values on the stack will be left as such. Even for non-symbolic values backends should take the frame location as a hint, and attempt to keep the value in a machine register
`mkPar`	number, number, [`fpParam`].	This instruction causes the value on the top of the stack is to be popped and transferred to the parameter location. It takes two arguments. The first of these is the size of the parameter, rounded to the smallest parameter size allowed in the machine conventions. The second is the parameter offset in the assembly area, or zero. The optional marker indicates that this is a floating point parameter
`mod`	mode.....	Takes the modulus of the two values on the top of the stack with semantics as specified by mode. Right operand is on top. The stack is popped by two, and the result pushed
`mul`	[mode] ...	Multiplies the two values on the top of the stack with semantics as specified by mode. The stack is popped by two, and the result pushed
`mulDbl`	....	Multiplies the two *double*s on the top of stack. The stack is popped by two, and the result pushed
`mulFlt`	....	Multiplies the two *float*s on the top of stack. The stack is popped by two, and the result pushed
`negate`	[mode] ...	Negates the integer on the top of stack. If mode is *intOver* then an attempt to negate the minimum integer is trapped. The result replaces the top of stack
`negDbl`	....	Negates the *double* on the top of stack. The top of stack is replaced by the result
`negFlt`	....	Negates the *float* on the top of stack. The top of stack is replaced by the result
`orWrd`	....	Bitwise *OR* of the two top of stack words. The stack is popped by two, and the result pushed
`pop1`	....	Pop the stack by one word
`popCall`	number...	Call the procedure whose address is on the top of stack with the specified number of parameters. The address is popped from the stack
`popFi`	number...	Pop the top of stack value into a location uniquely but arbitrarily associated with the ordinal number
`popRetD`	....	Pop the top of stack *double* into the double precision *double* return location
`popRetF`	....	Pop the top of stack *float* into the single precision *float* return location (see note 2)

opcode	params	description
`popRetW`	....	Pop the top of stack into the word return location
`pshAdr`	identifier .	Push the address of the statically allocated variable *identifier* onto the stack
`pshDsp`	index, offset	Push the address pointed to by the display vector element *display*[*index*] + *offset*.
`pshDstP`	....	This instruction pushes the contents of the dedicated location which holds the address of the return pointer. This is only used on architectures such as *SPARC* which do not pass destination pointers as an additional ordinary parameter
`pshFi`	number...	Push contents of the location uniquely but arbitrarily associated with the ordinal number onto the stack
`pshFP`	offset	Push the address at the given offset from the frame pointer $fp + *offset*
`pshLit`	number...	Push the literal onto the stack (see note 1)
`pshRetD`	....	Copy the *double* in the return location to the top of stack
`pshRetF`	....	Copy the *float* in the return location to the top of stack
`pshRetSB`	....	Copy the *signed byte* in the return location and sign-extend
`pshRetS16`	....	Copy the *signed 16-bit word* in the return location and sign-extend
`pshRetS32`	....	Copy the *signed 32-bit word* in the return location and sign-extend
`pshRetUB`	....	Copy the *unsigned byte* in the return location and zero-extend
`pshRetU16`	....	Copy the *unsigned 16-bit word* in the return location and zero-extend
`pshRetU32`	....	Copy the *unsigned 32-bit word* in the return location and zero-extend
`pshRetW`	....	Pushes the word return register to the top of stack
`pshTmp`	number...	Pushes the value associated with location $fp – *number* onto the stack. Matches mkTmp
`pshZ`	....	Pushes the literal 0 (zero)
`relEQ`	....	Compares the two top word-sized values. Returns *TRUE* if the second from the top is equal to the top. The stack is popped by two
`relNE`	....	Compares two top word-sized values. Returns *TRUE* if the second from the top not equal to the top. The stack is popped by two
`rem`	mode	Takes the remainder of the two values on the top of the stack with semantics as specified by mode. The stack is popped by two, and the result is pushed
`rotate`	....	Rotates the *word* second on stack left or right by the number of places specified by the top of stack value. Positive rotates are leftward. The stack is popped by two, and the result is pushed
`setExcl`	....	Excludes the bit specified by the value on the top of the stack from the word-sized bitset second on the stack. If a range test is required on the bit-specifying ordinal, this should be done explicitly. Implementations are free to perform the operation modulo-*bits-in-word*. The stack is popped by two, and the result pushed

opcode	**params**	**description**
`setIn`	....	Tests if the bit specified by the value on the top of the stack is set in the word-sized bitset second on the stack. Returns a Boolean on the top of stack. Top of stack word is taken modulo-*bits-in-word*. The stack is popped by two, and the result pushed
`setIncl`	....	Includes the bit specified by the value on the top of the stack from the word-sized bitset second on the stack. If a range test is required on the bit-specifying ordinal, this should be done explicitly. Implementations are free to perform the operation modulo-*bits-in-word*. The stack is popped by two, and the result pushed
`setGE`	....	Compares the two top values on the stack. Returns *TRUE* if the set second from the top is a superset of the top. The stack is popped by two, and the result pushed
`setLE`	....	Compares the two top values on the stack. Returns *TRUE* if the set second from the top is a subset of the top. The stack is popped by two, and the result pushed
`shLeft`	....	Shifts the value second on the stack left by the number of places specified by the unsigned value on the top of stack. The stack is popped by two, and the result pushed
`shiftV`	....	Shifts the value second on the stack left or right by the number of places specified by the integer value on the top of stack. Positive shifts are leftward. The stack is popped by two, and the result pushed
`shRightS`	....	Shifts the signed value second on the stack right by the number of places specified by the unsigned value on the top of stack. This is an arithmetic shift. The stack is popped by two, and the result pushed
`shRightU`	....	Shifts the unsigned value second on the stack right by the number of places specified by the unsigned value on the top of stack. This is a logical shift. The stack is popped by two, and the result pushed
`slash`	mode.....	Divides the two values on the top of the stack with semantics as specified by mode. The stack is popped by two, and the result pushed
`sub`	[mode] ...	Subtracts the two values on the top of the stack with overflow trapping semantics as specified by mode. The right operand is on top. The stack is popped by two, and the result pushed
`subDbl`	....	Subtract the two *double*s on the top of stack. The stack is popped by two, and the result pushed
`subFlt`	....	Subtract the two *float*s on the top of stack. The stack is popped by two, and the result pushed
`swap`	....	Swaps the two top values on the stack
`test`	identifier, number, number...	Perform the specified test-and-trap operation on the top of stack datum, then pop the stack

opcode	params	description
`trap`	identifier, number...	Same as `call` except no return is expected
`uToDbl`	....	Floats the top of stack from unsigned to *double* (*LFLOAT*). The top of stack is replaced by the result
`uToFlt`	....	Floats the unsigned value on the top of stack to *float* (*SFLOAT*) The top of stack is replaced by the result
`xorWrd`	....	Bitwise exclusive *OR* of the two top values. The stack is popped by two, and the result pushed

Note 1

Current backends assume that floating point instructions never have operands that have been pushed on the stack as literals.

Note 2

For machines which use an integer register for returning single precision floating point values *popRetF* and *popRetW* will be treated as synonyms.

1.6 Limitations on the control flow

DCode places the following restrictions on the control flow permitted in program representations.

- all control flow paths leading to any label in the code must have the same stack height along all edges leading to the label
- all control transfers, whether conditional or unconditional, must jump forward, except for back-edges of loops. All the back-edges of any particular loop must be enclosed in a *.LOOP, .ENDLOOP* directive pair
- the loop marker directives of nested loops must be properly nested

Backends which perform global dataflow analysis may very well not need this information.

Chapter 2

Rationale

2.1 Introduction

The notion of separating a compilation task into separate frontend and backend has a number of advantages. Firstly, it decouples the two tasks so that the same backend may be used for several language processors on the same machine, and a frontend may produce intermediate language for a variety of different targets.

So far as possible, the characteristics of the real target machine are hidden from the frontend processors which produce *DCode* but it is not possible to do this completely. In general, frontend language processors will need to know certain characteristic information. A typical set is given in appendix A.2. In the **gardens point** compilers such as **gp2d**, these characteristics are read from a configuration file named `gp2d.cfg`. These front ends produce files with the filename extension `.dcf`. The existing code generators may all be invoked by a base name to which they will append this default extension.

It is intended that once particular frontends are thoroughly tested the *ASCII* file form may be removed and a procedural interface used instead. The relationship between the compiler with a file interface for the intermediate representation and the final compiler with a procedural interface is shown in figure 2.1.

It should be noted that there are some costs inherent in this approach also. Chiefly, the "narrow" interface between the two components makes it difficult to use the richer set of operators which some machines provide. For example there is no provision in *DCode* for performing comparisons on data shorter than the natural word-size of the machine. Thus on target machines which support such operations the possibility must be foregone, or attribute information logically available to the frontend must be reconstructed in the backend.

2.1.1 Implicit assumptions

Current backends for *DCode* all make the following assumptions—

- programs will be linked to a runtime system which supplies certain global symbols needed for finding the display, checking the stack and so on. A typical set is given in appendix A.

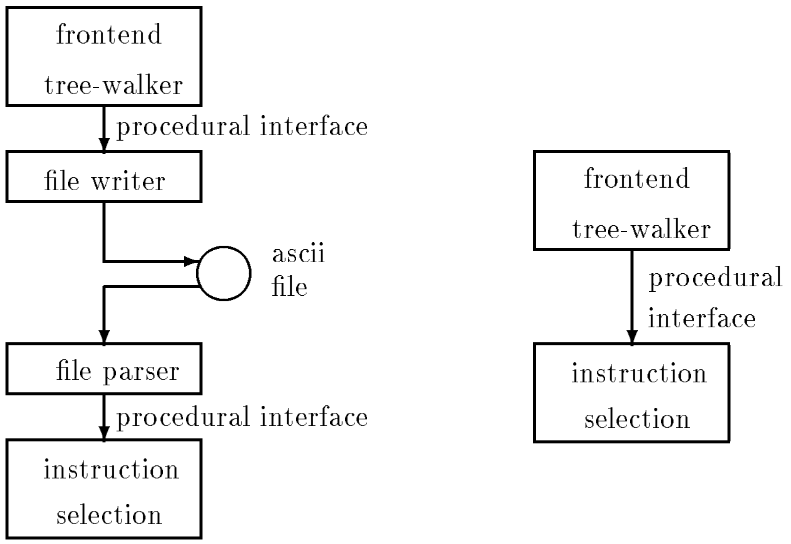

Figure 2.1: Compiler with file interface, and with procedural interface.

- any uplevel addressing is performed by means of a *display vector*. The automatically produced procedure prologs and epilogs save and restore the top of display as required. The changes which would be necessary, in order to use a static link mechanism are discussed in section 2.11.1
- it is assumed that the dereferencing of any address leads to a sign-extended (or zero-extended, as specified) value to be placed on the abstract stack. The machine does not perform short operations. Back-ends are free to shorten operations when they are able to detect that is correct (and efficient) to do so
- the method of passing *callee copies* open array values is fixed by the semantics of the *.OPENCOPY* directive of the abstract machine. A reference to the actual parameter appears first, followed by the upper array bound (*HIGH*) values for each of the dimensions in turn. The bounds may occur in any consistent order chosen by the frontend for a particular language.
- when the semantics of a language provides for value open arrays for which the element size can vary between the the caller and callee, value marshalling must take place in the caller. In such cases the caller's code must include specific *DCode* to marshall the value. In the unusual case where a procedure receives an open array and marshalls it before sending it on to another call, the code must dynamically allocate additional stack space with the *.EXPAND* directive

Back-ends do *not* make any assumptions about such things as the order of parameter evaluation, or the method of passing structured value parameters. The mechanism of transformation of an actual parameter value to the concrete stack or parameter register is abstracted away behind the `mkPar` pseudo-instruction. Code generators will generate the code to perform whatever (real) machine-level operations are required for this step. However, backends do not reorder the sequence of `mkPar`s in the intermediate representation. Thus it is the responsibility of the frontend to ensure that parameters are evaluated in an order which is appropriate for the target machine conventions.

The position of formal parameters and local variables in the stack frame is also left to the frontend to decide. For machines which push parameters onto the concrete stack this also affects the order of parameter evaluation. As a matter of choice **gp2d** uses the same conventions as language *C* compilers on the same platforms, to make mixed language programming easier. However, this is not a requirement of the *DCode*.

Similarly, backends are indifferent to the manner of passing of structured parameters of value mode. Back-ends will happily copy any parameter which appears in a *.COPY* directive, and conversely will work correctly with frontend processors which copy value arrays in the calling procedure rather than the more common *callee copies* convention. Again, **gp2d** follows the conventions of the native *C* compiler on the same machine.

2.1.2 DCode

DCode is an intermediate representation for computer programs which specifies instructions for an abstract stack machine. It is thus similar in its intent to P-Code[10, 1] and U-Code[8]. However, it differs from both of these in having a simplified model of memory access. *DCode* does not have any *load* instruction, but rather has *push-address* and *dereference* instructions. This avoids biasing the representation in favor of machines with a particular set of addressing modes, and reveals the whole of the address value computation. It should be remembered that for many of the more recent machines the loading of a base address will require two or more machine cycles. Thus a base address is potentially a valuable common subexpression, rather than a primitive which is repeatedly folded into the address fields of instructions. The abstract machine is thus "*RISC-like*."

Just as in U-Code, the representation contains a non-destructive store operation from the top of the abstract stack. This *make-temporary* and a matching *push-temporary* instruction form a basis for handling anonymous temporaries. The **gp2d** frontend uses this facility for the translation of Modula's *WITH* and *FOR* statements. Tools which perform common subexpression elimination may create many more temporaries of this kind. It should be noted that the generation of such temporaries in the frontend may place restrictions on the control flow graphs which are safe. This point is discussed further in the section 2.8.

The representation was not designed with any particular source language in mind, but most experience with the representation has been with Modula-2. A similar representation has been used as the basis for a prototype C++ compiler, and *DCode* is also being used for Oberon-2. *DCode* has been used for direct interpretation, in the **gpm-pc** implementation[7]. A *DCode* backend for Fraser and Hanson's `lcc`[5] has also been produced.

Values on the abstract stack are typed. Nevertheless, operations such as `add` exist in different forms for integers, single and double precision floating point. The arithmetic operators are neither parameterized by type nor are they polymorphic, although either of these choices would be logically possible. This design choice was made to reduce the number of attributes known to the frontend which need to be reevaluated in the backend. In the case of fixed point arithmetic, the overflow trapping mode, if any, is explicitly specified.

2.2 The D-Code format

2.2.1 A simple example

Figure 2.2 is an example of a simple Modula-2 program, and figure 2.3 is the output of **gp2d**.

```
MODULE Hello;
  FROM ProgArgs IMPORT Assert, ArgNumber;
  FROM InOut IMPORT WriteString, WriteCard, WriteLn;

BEGIN
  WriteString("Hello, world"); WriteLn;
  Assert(ArgNumber() = 1,"hello needs no args");
END Hello.
```

Figure 2.2: Hello world in Modula-2.

The *DCode* file in figure 2.3 is exactly as produced by **gp2d** except for the comments which follow the semicolons in the instruction sequence.

In this figure **gp2d** has folded all of the literal information into a single array of bytes *cDat*. The information following the semicolon on each line is a comment, and gives the *ASCII* representation of the constant data. In the case of hand-written files it would be more normal to declare such data with an *.ASCIIZ* directive. The "procedure" *Assert* does not appear in the import list of the *DCode* file, because this is a system procedure in this Modula-2 implementation, and is expanded inline in the code.

In general, the format uses directive keywords which start with a fullstop and in many cases have similar meanings to the equivalent Modula keywords.

2.3 Lexical issues

The lexis of *DCode* is set out in section 1.2.

Starting with version 2.0 it is again possible to include files verbatim in the *DCode* file. The format is —

`#include` *literal-string*

Only one level of inclusion is permitted.

This facility is intended for the use of frontends for languages such as *C*, which are inherently unable to emit import and export lists until they have read all of the source file. The idea is to write the import and export list to a separate file, and #include that in the *DCode* file at the appropriate place.

```
; D-Code output produced by gpm from "hello.mod"
.TITLE Hello
.FILE "hello.mod"

.EXPORT _StartHello
.IMPORT _InOut_WriteLn
.IMPORT _ProgArgs_ArgNumber
.IMPORT _InOut_WriteCard
.IMPORT _InOut_WriteString

.CONST
__mNam: .ASCIIZ "Hello"
.CONST
__cDat: .BYTE 048H,065H,06CH,06CH,06FH,02CH,020H,077H   ; [Hello, w]
        .BYTE 06FH,072H,06CH,064H,000H,068H,065H,06CH   ; [orld.hel]
        .BYTE 06CH,06FH,020H,06EH,065H,065H,064H,073H   ; [lo needs]
        .BYTE 020H,06EH,06FH,020H,061H,072H,067H,073H   ; [ no args]
        .BYTE 000H                                      ; [.]

.PROC _StartHello(.NOCHECK,.SIZE=0,.NODISPLAY)
.ENTRY
lineNum 6
        pshLit 12                           ; push the high value
        mkPar 4,4                           ; parameter #2
        pshAdr __cDat                       ; const string
        mkPar 4,0                           ; parameter #1
        call _InOut_WriteString,2           ; call procedure
        cutPars 8                           ; remove params
lineNum 7
        call _InOut_WriteLn,0               ; call procedure
        .TRAP __gp_assTrp,label1,__cDat+13,0 ; declare trap-label
        call _ProgArgs_ArgNumber,0          ; call function
        pshRetW                             ; push returned value
        pshLit 1                            ; literal 1 value
        relEQ                               ; compare equal, if false
        brFalse label1                      ; branch to trap label,
        exit                                ; else just return
        endF
.ENDP
```

Figure 2.3: Hello world in *DCode*.

2.3.1 Comment pragmas

In version 2.01 a special comment format has been introduced which allows frontends to pass arbitrary information to the backend output file. Such comments begin with an exclamation point. Here is an example —

```
;! .stabs "__gpm_compiled.", 0x3c, 0, 0, 0
```

Backends which support such pragmas will simply copy the remainder of the line to their ouput files.

In this example, the intent is to pass symbol table information from the frontend to the assembler. Note however, that this implies an undesirable coupling between the frontend and the target. In this case, the frontend not only is parameterised for the target, but must also understand the details of the assembler file format. Currently in **gp2d** this facility is only being used in versions which use the Free Software Foundation's `gdb` debugger. Further information is given in the section 2.10.4.

2.3.2 Primitive data objects

The primitive data types known to *DCode* are *unsigned byte*, *signed byte*, *unsigned 16-bit word*, *signed 16-bit word*, *unsigned 32-bit word*, *signed 32-bit word*, *word*, *float*, and *double*. The *word* type is the "natural" word size of the machine, while other types are of fixed sizes and may be shorter. In certain machines, several of these types may be treated as synonyms by the backends.

In 32-bit machines it would be typical for *unsigned 32-bit word*, *signed 32-bit word* and *word* to all be the 32-bit data type. In 64-bit machines *unsigned 32-bit word* and *signed 32-bit word* would be the unsigned and signed 32-bit data types, while *word* would be the full 64-bits.

There are obsolescent architectures for which this particular design does not easily fit. However, portablility to such machines poses other problems beyond those addressed by this intermediate form.

2.3.3 Declarations

The declarations part of the file begins with imports and exports. Exported names are visible to the linker, while imported names are expected to be resolved to locations external to the module.

The title declaration has no particular function, except for identification purposes for the human reader. In contrast, the *.FILE* declaration is assumed to be the name of the source file, and will be written to the object file by all backends, for the benefit of debuggers.

Constant declarations may be of bytes, various short words, and words, or may be character strings. String declarations are of the following form —

```
.CONST 13
   str: .ASCIIZ "Hello, world"
```

This declaration appends a zero byte at the end of the string, while the corresponding *.ASCII* declaration does not.

Some compilers, such as the vendor-supplied compilers with *MIPS* machines, place data in different segments depending on the size of the object. The optional number following the keyword *.CONST* in the *DCode* is the size of the following block. This allows *DCode* backends to place data using the same strategy, thus enhancing compatability with object code produced by other compilers.

Each new group of declared constants starts with a *.CONST* symbol, and at least one label. All labels are placed in memory on the most stringent alignment boundary. Thus in figure 2.3 the label `__cDat` is aligned on a boundary which does not depend on the length of the `__mNam` string. This semantic rule allows frontends to declare initialized data, and be assured that the alignment of the data depends only on the offset from the label. In general, backends must ensure that initialized data corresponding to any single *CONST* declaration remain contiguous and ordered in the memory space, except for any padding necessary to place each label in the group on a sufficiently stringent boundary.

Backends are free to allocate space for *CONST* objects in read only memory segments, for those machines which support such segmentation. Thus initialized *variables* must be declared in *.DATA* declarations. The syntax of the data declarations is otherwise identical to constant declarations. Backends are obliged to allocate space in the data segment for objects appearing in *.DATA* declarations. For machines with multiple data segements, backends are free to place data from different declaration groups in different segments according to storage size. The optional size number facilitates this strategy.

Declarations of uninitialized variables declare names in the *BSS* segment, and specify the storage size in the declaration.As an example, an array of *REAL* of three elements named *coords* could be declared in any of the following ways —

```
.VAR
   _coords: .BYTE   24          ; 24  bytes
   _coords: .BITS32 6           ; 6 * 4-byte words
   _coords: .DOUBLE 3           ; 3 * 8-byte doubles
```

Of course the size of the element should correspond to the alignment which the target machine requires. **gp2d** always aligns such objects correctly, taking into account the alignment rules in the file *gp2d.cfg*.

The optional *.ENTRY* field in variable declarations allows for labels to be declared at the stated byte offset from the start of the memory space allocated for the variable. For example, the declaration —

```
.VAR
   _objectN: .WORD 7 .ENTRY 8
```

declares a variable of size seven words, with the label placed at an offset eight bytes from the start of the 28 byte block. All backends will ensure that the eight bytes before the labelled address are contiguous with the 20 bytes beyond the address. In some assemblers this may require the declaration of a dummy label for the entire variable, and the automatic translation of references to *objectN* into *dummy+8*. Of course, any such contortions are hidden from the *DCode* interface.

2.3.4 Procedure headers

Procedure declarations begin with the marker keyword *.PROC* and end with the keyword *.ENDP*. The header has the declared name of the procedure (which may or may not correspond to an exported name) and has entry information. The *.PROC* keyword may be preceded by the optional keyword *.LOCAL*. If present, this marker states that the procedure is neither exported or assigned as a procedure variable. In some architectures this allows some call optimizations to be made.

The keywords *.CHECK, .NOCHECK* in the procedure header indicate whether or not a stack overflow check is to be performed. Checking is the default, so the check keyword is usually included simply for clarity.

The keywords *.DISPLAY, .NODISPLAY* indicate whether or not a display location needs to be updated. In the case that the display is needed the level is indicated by the notation—

`.DISPLAY :` *lexLevel*

Compilation unit bodies are at lexical level 0, and require no display. The default is no display, which in turn is the same as display at level zero. If a procedure has no objects which are accessed from nested scopes, then no display updating is required, and it is permissible to declare *.NODISPLAY*.

The *.SIZE = number* phrase declares the size of the (fixed part of the) stackframe. The number following the equality is the size in bytes. The default size is zero. The size includes any callee copied structures of fixed size, but does not include conformant arrays.

The *.ASSEMBLY = number* phrase declares the size of the parameter assembly area in the stack frame. This is used only in configurations which have declared *fixedParams*. The number following the equality is the size in bytes. The default size is zero. Parameter passing conventions are explained below.

2.3.5 Copy and Expand declarations

The procedure parameter conventions may provide that structured parameters of value mode are copied by the called procedure. There are two separate declarations for this purpose. Fixed size value structures are declared by the line —

`.COPY` *paramOffset , destOffset* `.SIZE` *objectSize* `.ALIGN` *alignNum*

With the usual conventions for the handling of the runtime stack, parameters are at higher addresses than the return address, and hence have positive offsets from the frame pointer. The *paramOffset* value is thus always positive. The destination for the copied structure is in the local variable part of the stack frame (see the diagram B.1 in the appendix B) and hence *destOffset* is always negative. In the case of runtime stacks which grow upward in the address space the signs are reversed. The *objectSize* is measured in bytes, and *alignNum* is the alignment of the actual which is to be copied.

The copying of value variable size, open arrays is indicated as follows —

`.OPENCOPY(`*dimensions*`)` *paramOffset* `.SIZE` *elementSize* `.ALIGN` *alignNum*

In this case, the destination offset is determined at runtime. The size of the array copy is determined at runtime from the given element size, and the array *HIGH* value(s). The high values are found at[1] $paramOffset + ptrSize$, $paramOffset + 2 \times ptrSize$ and so on. The optional number in parenthesis states the dimensionality of the array. The default is one.

After performing an open array copy, the parameter should be updated to point to the copy. Alignment of the new stack-top is the responsibility of the code which copies the open array.

Some languages (notably draft standard Modula-2) provide that values may be passed to open array parameters, provided that the actual and formal element types are *assignment compatible*. Such semantics would allow the element sizes of the actual array to be different to the formal array. In such a case the value copy must be made in the caller, so that any widening or narrowing (together with a per-element bounds test) is done during this marshalling.

In almost all cases the size of the actual is known to the caller at compile time, so that the frontend can allocate space in the stack frame for the copy. A normal `blkCp` operation suffices for simple cases, while a loop in the *DCode* is necessary if any marshalling or element-by-element range checking is required.

However, if a procedure receives an open array, and passes it on to another procedure, then the size of the copy is not known until runtime. In such cases code in the procedure prolog must compute the size of the copy area, and allocate additional space in the stack frame.[2] It is necessary to have a pointer at a known location which will point to the allocated space. The format is —

`.EXPAND`(*dimensions*) *paramOfst, dstPtrOfst* `.SIZE` *elemSize* `.ALIGN` *alignNum*

As before, the number of dimensions is optional, with a default of 1. The frame offset of the pointer to the actual is given by *paramOfst*. The frame offsets of the high values are found from this value, as for open arrays. The stack frame location at which the pointer to the allocated space is deposited by the prolog is given by *dstPtrOfst*, while *elemSize* is the size of the formal element size to which the actual array is marshalled. As usual, *alignNum* is the alignment of the actual parameter array.

It is possible that the same incoming formal open array might be marshalled and sent out to several procedures. In such a case the stack frame expansion area may be shared between the calls, provided that the element size is declared equal to the largest outgoing formal element size.

A different pathological case arises if the same incoming open array is passed to several formals in the same call. In such a case multiple *.EXPAND* declarations must point to the same incoming parameter offset.

The manipulations involved in marshalling an open array will assume that the procedure prolog has allocated space using the *.EXPAND* directive, and that the stack offset *dstPtrOfst* points to the allocated space. The calling procedure must assemble the parameter copy into the expansion area, performing any range checking, widening or narrowing element-by-element. The value at *dstPtrOfst* is then pushed onto the abstract machine stack and made into a reference parameter by an appropriate `mkPar` instruction.

[1] In the rare case of machines with the *stackUpright* declaration the location would be $paramOffset - ptrSize$, and so on.

[2] In the case of procedure conventions which must have fixed stack frame sizes it is necessary for the prolog to allocate space using *soap*, the separate stack for open array parameters. In this case, the procedure epilog must release the space back to the runtime system at procedure exit.

2.3.6 Local information

The local variable information is indicated by the *.LOCAL* keyword. It is included for the information of human reader, and to simplify the construction of code-optimizer programs.

WARNING – many RISC machine code generators require local declarations, at least for formal parameters of procedures. Unless this information is given, the backend may not be aware that values have been placed in the parameter registers. Likewise, the optional "fpParam" flag may be needed to ensure that RISC backends know that the parameter is a floating point value, and thus may be in a floating point register. The necessity to declare such parameters extends to the anonymous parameter which points to the destination location for functions returning structured types, and to the reference which is passed to copy and open copy parameters.

The format of the local declaration is as follows —

`.LOCAL` *ident offset , size* (*uplevAccess,uplevThreat,adrTaken*)

The three arguments in the parentheses are either zero or one, and are used by code-optimizer backends. They have the following meanings, where in each case the true value for the attribute is indicated by an argument of one.

The first indicates whether or not the object is uplevel accessed through the display. The second indicates whether or not the object is threatened by an uplevel access, that is, whether it is assigned to, or passed as a variable mode parameter by a nested procedure. The final attribute indicates whether or not the object must reside in memory because it has its address taken. This is the case where the address is taken directly, or by being passed as actual parameter to a variable mode formal parameter, or by being used as the base address of an address computation. Because of this final possibility, in general, the third attribute should have the true value if the local object is a structure or array.

In the case that simple frontends do not evaluate these Boolean attributes it is always conservative and safe to declare (1,1,1). In the case of languages without uplevel addressing the conservative choice would be (0,0,1).

For the case of machines which have the *fixedParams* declaration integer and floating point parameters may be passed in different register sets. Thus it is necessary for the backend to know if a particular parameter is a floating point value or not. For the floating point case, the local declaration should have the form —

`.LOCAL` *ident offset , size* (*bool1,bool2,bool3*) `fpParam`

Local declarations may be followed by a literal string. This literal string is used in some versions of **gp2d** to pass type information for debuggers. For further information, see the section 2.10.4

2.3.7 Pascal call conventions

Procedures which use the so-called Pascal call conventions, require each procedure to remove its own parameters from the stack during the procedure epilog. Such procedures need a marker in the header of the form —

`.RETCUT =` *number*

2.3.8 Exception handling primitives

As a minimal construct for the building of exception handling models with low runtime overheads, label tags can be placed on labels which are entry points of exceptions handlers, and retry points for exception handling mechanisms which support retrial semantics. The optional exception entry point tag is of the form —

`.EXCEPT` *label*:

This declares that the alternative entry point after the catching of an exception in the current procedure, or its callees, is the label given in the label. The underlying exception handling model is low-level, but perhaps flexible enough to be a basis for different user-level semantics.

When entering a procedure at the normal entry point, if the procedure has an exception label declared, an exception handler is “set”. If an exception is subsequently raised, and this handler is the most current, then the stack is cut back until the same procedure frame is on top of the stack once more. The handler code is then entered at the specified label. Backends which support this feature are expected to insert appropriate code into the procedure prolog to store the exception label address in whatever data structure the runtime support and trap handler require.

There are a number of ways of implementing the functionality specified here. These may be of lesser or greater runtime overhead. For example, as an extreme case the procedure prolog might insert code which calls the standard *C* library function `setjmp` along with code which dispatches error returns to the exception entry label.

For exception handling mechanisms which provid for retrial semantics, it is necessary for the frontend to insert a marker on the label to which control is transferred for such retrial. Note that it is not convenient for the backend to deduce the position of such a label, since the detailed semantics of the language determine which parts, if any, of the procedure entry prolog should be repeated in a retrial. The format of the declaration is —

`.RETRY` *label*:

In this case, the region protected by the exception handler is held to be entered when control first reaches the nominated label.

2.4 The D-Code instruction set

The instruction set is divided into two parts, the basic instruction set, and the floating point instructions. There are a few extra instructions which are only necessary for some machines, such as the instructions which deal with variable size return registers, and segmented addressing.

A full description of the instructions, in alphabetical order is given in section 1.5.2. The following tables show the instructions grouped according to type. In these tables, a parameter in square brackets is optional.

2.5 Trapping modes

The basic integer instructions

opcode	params	opcode	params
unary operations			
`abs`	[mode]	`bitNeg`	—
`boolNeg`	—	`flatten`	—
`makeAdr`	—	`negate`	[mode]
binary operations			
`add`	[mode]	`addAdr`	—
`addOff`	offset	`andWrd`	—
`div`	mode	`mod`	mode
`mul`	[mode]	`orWrd`	—
`rem`	mode	`rotate`	—
`setExcl`	—	`setIn`	—
`setIncl`	—	`shiftV`	—
`shLeft`	—	`shRightS`	—
`shRightU`	—	`slash`	mode
`sub`	[mode]	`xorWrd`	—
load and store operations			
`assignB`	—	`assign16`	—
`assign32`	—	`assignW`	—
`derefSB`	—	`derefS16`	—
`derefS32`	—	`derefUB`	—
`derefU16`	—	`derefU32`	—
`derefW`	—	`pshAdr`	identifier
`pshDsp`	index,offset	`pshFP`	offset
`pshLit`	number	`pshZ`	—
flow of control operations			
`brFalse`	label	`brTrue`	label
`branch`	label	`call`	identifier,number
`jump`	—	`popCall`	number
`trap`	identifier,number		
relational operations			
`crdGE`	—	`crdGT`	—
`crdLE`	—	`crdLS`	—
`intGE`	—	`intGT`	—
`intLE`	—	`intLS`	—
`relEQ`	—	`relNE`	—
`setGE`	—	`setLE`	—
function return value operations			
`popRetW`	—	`pshRetSB`	—
`pshRetS16`	—	`pshRetS32`	—
`pshRetUB`	—	`pshRetU16`	—
`pshRetU32`	—	`pshRetW`	—
parameter operations			
`blkPar`	size,offset[,align]	`cutPars`	number
`mkDstP`	size	`mkPar`	size,offset[,`fpParam`]
`pshDstP`	—		

opcode	params	opcode	params
miscellaneous operations			
`blkCp`	[align-number]	`dup1`	—
`endF`	—	`endP`	—
`exit`	—	`lineNum`	number
`mkTmp`	number	`pop1`	—
`popFi`	number	`pshFi`	number
`swap`	—	`test`	testId,param ...
`pshTmp`	number		

Floating point instructions

opcode	params	opcode	params
floating point unary operations			
`absDbl`	—	`absFlt`	—
`negDbl`	—	`negFlt`	—
floating point binary operations			
`addDbl`	—	`addFlt`	—
`divDbl`	—	`divFlt`	—
`mulDbl`	—	`mulFlt`	—
`subDbl`	—	`subFlt`	—
floating point load and store operations			
`assignD`	—	`assignF`	—
`derefF`	—	`derefD`	—
floating point conversion operations			
`dFloor`	[mode]	`fFloor`	[mode]
`dRound`	[mode]	`fRound`	[mode]
`dTrunc`	[mode]	`fTrunc`	[mode]
`dToFlt`	—	`fToDbl`	—
`iToDbl`	—	`iToFlt`	—
`uToDbl`	—	`uToFlt`	—
floating point relational operations			
`dblRel`	relop	`fltRel`	relop
floating point function return operations			
`popRetD`	—	`popRetF`	—
`pshRetD`	—	`pshRetF`	—

Many instructions have a either a mandatory or optional trapping mode specified. These modes are one of *noTrap, crdOver, intOver*, corresponding no trapping, unsigned overflow or signed overflow respectively. However, not all modes apply to all instructions, and the details in the alphabetical listing at the back of this report should always be checked.

In the case of the dividing and remaindering operators the mode is mandatory, since the signedness of the operands is part of the semantics, even for non-overflowing cases.

In the case of floating-point to whole-number conversions, there are three different conversions. Floating point numbers may be converted with rounding to nearest, rounding toward minus infinity, and truncation toward zero. Conversions may be untested, or tested for either signed or unsigned overflow. Backends must ensure that untested conversions give correct results in both the integer and unsigned ranges.

2.6 Memory Addressing

Memory is arbitrarily divided into four categories. These are static addresses, local addresses, uplevel addresses (for those languages which allow nested procedure definitions) and pointer addresses.

2.6.1 Pushing addresses

Static addresses

Static addresses are those which occupy fixed locations throughout the execution of the procedure. These are accessed by their symbolic names in the *DCode*. The format is —

`pshAdr` *identifier* [± *number*]

This instruction pushes the address of the nominated datum onto the stack of the abstract machine.

The optional offset number allows the frontend to fold constant offsets, as might occur in the access of a record field, or an array element at a fixed index.

Stack frame addresses

Local addresses are those whose location in the current stack frame is known at compile time. These are accessed by means of an offset from the abstract *frame pointer* (*FP*). The format is —

`pshFP` *number*

This instruction pushes the address of the nominated datum onto the stack of the abstract machine.

In this case the number is a signed offset. In the usual case of stacks which grow downward in memory space, positive offsets access the parameters, while negative offsets access local variables and temporaries.

Uplevel addresses

Uplevel addresses are those whose location is known relative to the the stack frame pointer of a statically (and hence dynamically) enclosing procedure. It is assumed that the value of this frame pointer is stored in a *display vector*. In *DCode*, the format used to push the address of a datum at offset *offset-number* from the stack frame pointer value stored at lexical level *lexical-level-number* in the display is —

`pshDsp` *lexical-level-number, offset-number*

Since we are using a display vector here, the absolute nesting level of the target procedure is required. In conventions which use static links, it is the *difference* in lexical nesting level which is important. The use of static links with *DCode* is discussed in section 2.11.1. In such cases, the `pshDsp` instruction is not used.

Pointer addresses

If the address of a pointer is on the top of the abstract stack, the address of the bound object is obtained by dereferencing the word on the top of stack.

2.6.2 Dereferencing data

Values are pushed onto the abstract stack by using an appropriate dereference instruction. There are nine of these, one each for the nine primitive data types known to *DCode*. These are *unsigned byte*, *signed byte*, *unsigned 16-bit word*, *signed 16-bit word*, *unsigned 32-bit word*, *signed 32-bit word*, *word*, *float*, and *double*. The dereference instruction replaces the top-of-stack address by the datum which it points to. If the address is still needed, it may be copied prior to the deref operation.

Here is a typical idiom which is used to increment a memory location by one. It is assumed (to make the example more interesting) that the location is pointed to by a reference (*VAR*) parameter, at stack location \$fp + 8.

```
pshFP   8       ; push address of parameter
derefW          ; now we have the address of the actual parameter
dup1            ; now we have two copies of the address
derefW          ; dereference the top copy of the address
pshLit  1       ; add one to this original value next ...
add             ; memory address is still next on stack
swap            ; assign wants address on top of stack
assignW         ; store result
```

Note that by using the `dup1`, the task of computing the address is performed only once, no matter how complex the address computation is.

2.6.3 Assigning values

Values are assigned to memory using the various assign instructions. The (right-hand-side) value is first pushed onto the abstract stack, then the destination (left-hand-side) address is pushed, and the assign instruction emitted. This does not mean that the two data have to be evaluated in that order. As demonstrated in the preceding example, it is possible to compute the data in the opposite order and then swap the positions on the abstract stack.

Front-ends which apply the Sethi-Ullman algorithm[9] to determine an optimal order of evaluation will first compute whichever side of an assignment statement requires the deepest stack. If this is the left hand side of the assignment, then a `swap` will be emitted immediately before the `assign`*X*.

Assignment of entire values is done by means of a block copy instruction. In this case, the destination address is pushed first, the source address next, and the size of the block (in bytes) last. For some machines it is advantageous to know the alignment of the two addresses, since it is always more efficient to copy several bytes at a time, if the alignment constraints of the target machine allow this.

Here is an example which copies a local structure of size 16 bytes, at offset −24 in the stack frame, to the static location named *globalStruct*.

```
pshAdr  _globalStruct  ; destination address is pushed
pshFP   -24            ; source address in local frame
pshLit  16             ; size of the block in bytes
blkCp   4              ; quad-aligned, so ok to copy word-by-word
```

2.7 The stack frame format

The stack frame format may be almost freely decided by the frontend processor. However, some restrictions apply. Within limits determined by the hardware requirements of the target machine, frontends may place local variables at any location in the stack frame. A detailed description of the conventions used by **gp2d** are set out in the appendix B.

The examples in this section will assume that the stack grows downward in the address space.

In general, a stack frame consists of four parts (see figure 2.4). The *incoming parameters*, the *stack mark*, the *local variables* and any memory temporaries which are allocated either by the frontend or the code generator.

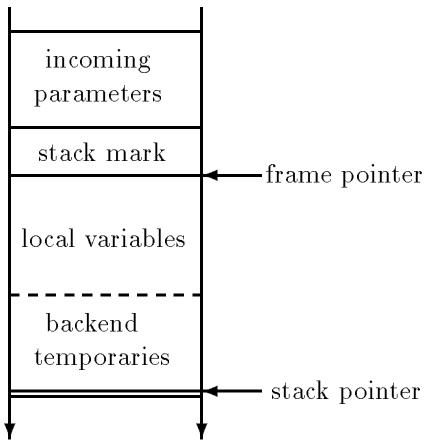

Figure 2.4: Generic stack frame, (stack grows downward).

In principle, a register (the *frame pointer* or the *local pointer*) points to the stack mark, which is situated between the parameters and the local variables. The calling procedure places the parameters in the parameter area, and then calls the machine's procedure call instruction. The stack mark contains the information which is needed for the procedure to return correctly. This is typically two data — the old frame pointer value, and the return address. In some machines the creation of the stack mark is performed by the call instruction, in more recent machines the steps are explicitly programmed as part of the procedure entry prolog. In the case of machines which require an area for the automatic saving of register windows, the runtime stack frame size will need to be adjusted by the code generator. This is probably best done by incorporating this extra requirement into the abstraction of the "stack mark".

Space for local variables is allocated by subtracting a constant from the stack pointer value. The size is the space required for all the declared local variables, plus any space required for anonymous temporary values, plus any space required to save any *callee saves* registers.

For most languages, the size of this region is known at compile time, although possibly only after register allocation is complete. For languages which have open or conformant array parameters which are copied by the callee,the size of the local variable space is not known until runtime.

It is convenient to allocate space in the stack frame in the order that the need becomes known. Thus declared local variables (the size of which is known in the frontend) should be allocated first, at known offsets. If the conventions require the callee to copy structured value parameters, the space to do this must be reckoned as part of this size, as must frontend allocated temporaries. Existing *DCode* backends will use the first few bytes of the frame to save the current display value, for those procedures which need to do this. This use moves all other offsets down by *displayElSize*, and must be taken into account by the frontend procedure which assigns offsets to local variables.

If the backend code generator needs to allocate temporaries this must be done after the area declared by the frontend, as the need becomes known only later in the compilation process. Back-ends must thus expand the space declared by the *SIZE* declaration in the procedure heading.

Finally, if the there are callee-copied open array parameters, space must be allocated at runtime, extending the stack frame still further.

2.7.1 Parameter passing conventions

There are two strategies for the assembly of parameters, and their passage to the called procedure. Actual parameters may be pushed on the stack as they are evaluated, or assembled in a fixed area on the top of the stack frame.

Parameters pushed on the runtime stack

Parameters are pushed on the stack as they are evaluated. For language C the first parameter is pushed last, so that it is on the "top" of the parameter stack. The first parameter is thus always found by the called procedure at offset ($\$fp + \mathit{stackMarkSize}$).

In the case of functions returning structured values, it is normal for the caller to allocate space for the return value, and to push a pointer to this value as an additional, anonymous parameter. The called procedure should copy the result to this location, and also return the pointer in the normal way (usually in a register).

Consider the following procedure —

```
PROCEDURE Position(now : Time) : CoordType;
(* CoordType = RECORD x, y, z : REAL END *)
```

The procedure call `WritePos(Position(currentTime));` might be a typical use of this function. The compiler allocates a 24-byte anonymous temporary within the current stack frame, to buffer the result between the return of `Position` and the calling of `WritePos`. We assume that this temporary is at location ($fp – 48). This call would be translated as follows —

```
pshAdr   _currentTime
derefW                  ; get the value parameter for Position()
mkPar    4,0            ; copy 4 bytes to top of runtime stack
pshFP    -48            ; push address of temp for return value
mkPar    4,0            ; copy 4 bytes to top of runtime stack
call     _Position,2 ; call the function
cutPars 8               ; remove params, 2 * 4 bytes
pshRetW                 ; psh address of result, same as temp address
blkPar   24,0           ; copy 24 bytes to top of runtime stack
call     _WritePos,1
cutPars 24              ; remove param, 24 bytes
```

In this example, the second parameter to the *xxx*`Par` instructions are not used. In this (stack parameter) case each parameter is always copied to the top of the runtime stack. However, the instruction syntax demands that a dummy value, such as zero, be included.

Inside the example procedure `Position`, the destination location would be accessed via the extra parameter. If a particular language frontend uses the convention that destination locations are always temporaries then the function may construct the result in place. Otherwise, the result must be returned by means of a block copy instruction. Here is an example, where the result has been constructed in the stack frame at location ($fp – 32), and where the stack mark size is 8.

```
...                 ; result is in fp-32
pshFP    8          ; dst pointer is at fp+8
derefW              ; get the destination address
pshFP    -32        ; push the source address
pshLit   24         ; push the block size
blkCp    4          ; copy to destination (word-wise)
exit
```

Parameters assembled in a fixed location

The stack frame has an additional area defined at the top of the frame, which is the parameter assembly area. This area must be as large as the parameters of the called procedure of greatest parameter size. It is thus a requirement of the frontend to evaluate this size, and pass it to the backend with the *.ASSEMBLY* declaration.

This method of assembling parameters has some advantages in efficiency for any machine which is able to address memory relative to the stack pointer. It is thus useful for Intel 80386, but not for earlier Intel *iapx*86 processors. All parameter passing conventions which pass some parameters in registers should use the fixed parameter assembly area strategy.

There is another case where the use of the fixed area is mandated. This arises when the alignment constraints of some parameter types are more stringent than the constraints of the runtime stack itself. For example, a machine which required octo-byte alignment for *doubles*, could not simply push a double value onto the runtime stack. Instead, it is simplest to assign appropriate offsets for each formal parameter, and place the actual parameters at these offsets in the assembly area. It may be noted that the offsets in the assembly area are not quite the same as the offsets which the called procedure will use. The first is the offset from the *stack pointer* of the caller, the second from the *frame pointer* of the callee.

Parameters cannot be evaluated in canonical order in the case of fixed assembly. Because the same area is used to assemble parameters for all of the procedures called by any particular procedure, use of the area must be serial.

Structured return values are handled by the creation of an additional, anonymous variable-mode parameter, just as for the *stackParams* case. The nested function call used as an example previously now becomes —

```
pshAdr  _currentTime
derefW              ; get the value parameter for Position()
mkPar   4,4         ; copy 4 bytes to (sp+4)
pshFP   -48         ; push address of temp for return value
mkPar   4,0         ; copy 4 bytes to (sp+0)
call    _Position,2 ; call the function
pshRetW             ; psh address of result, same as temp address
blkPar  24,0        ; copy 24 bytes starting at (sp+0)
call    _WritePos,1
```

In this second case the parameter copy instructions move the parameter on the top of the abstract stack to the parameter assembly area of the runtime stack, at the specified offset. No change to the runtime stack pointer is made. Since the parameter assembly area is allocated for the whole of the procedure activation, no removal of parameters following the calls is required. Note that the presence of the destination parameter will increase the assembly area offsets of all the other parameters.

Within the called procedure, as in the previous case, the destination location would be accessed via the extra parameter. Here is an example of returning the result, where the result has been constructed in the stack frame at location ($fp − 32), and where the stack mark size is 8.

```
...                 ; result is in fp-32
pshFP   8           ; dst pointer is at fp+8
derefW              ; get the destination address
pshFP   -32         ; push the source address
pshLit  24          ; push the block size
blkCp   4           ; copy to destination (word-wise)
exit
```

Within the *called* procedures, parameters are accessed via the *frame pointer*, while within the *calling* procedure the parameters are assembled at offsets which are relative to the *stack pointer*. These two offsets differ by the size of the stack mark.

The backend is expected to decide which parameter offsets (if any) will actually correspond to register locations. The frontend deals only with the abstraction of the assembly area.

Some conventions for passing destination pointers for functions returning structured values involve the use of a dedicated location in the stack frame. For example, SUN Microsystems conventions for *SPARC* processors have this property. *DCode* has a pair of instructions which deal with such locations without the frontend having to know the actual stack position. The `mkDstP` instruction causes the top of the abstract stack to be moved to the dedicated location. This mechanism should, of course, only be used for architectures which do **not** pass the destination pointer as a normal parameter.

Here is the same example, for the *SPARC* case.

```
pshAdr  _currentTime
derefW               ; get the value parameter for Position()
mkPar   4,0          ; copy 4 bytes to (sp+0)
pshFP   -48          ; push address of temp for return value
mkDstP  24           ; move ptr to 24-byte dest. to special location
call    _Position,1 ; call the function
pshRetW              ; psh address of result, same as temp address
blkPar  24,0         ; copy 24 bytes starting at (sp+0)
call    _WritePos,1
```

In this case, the return of the block result requires an access to the dedicated destination location, which is arranged using the special `pshDstP` instruction which matches the special `mkDstP`. Here is the code, for the case where the result has been constructed at location ($fp − 32).

```
...                  ; result is in fp-32
pshDstP              ; get pointer to destination
pshFP   -32          ; push the source address
pshLit  24           ; push the block size
blkCp   4            ; word-by-word copy to destination
exit
```

Machines without frame pointer registers

Some machines use conventions in which no register is allocated as a frame pointer. For example the *MIPS* compilers do not use a frame pointer register, but instead access all data in the stack frame by using offsets from the stack pointer. This technique saves a register for other uses, and also makes procedure calls slightly faster. Unfortunately, this technique demands that the stack frame size be known at compile time. Such a demand does not fit easily with languages which have open array parameters, since the size of these is not known at compile time.

There are essentially two possible solutions to this problem. One is to use a separate (*SOAP*) *S*tack for *O*pen *A*rray *P*arameters, the other is to ignore the convention and allocate another, dedicated register. Note however that this matter may be left to the backends to decide, and has no influence on the *DCode* abstraction.

In particular, it should be noted that the absence of a "real" frame pointer at runtime has no influence on the form of the *DCode* except insofar as it affects the computed offsets of parameters. Front-ends still use the abstraction of the frame pointer, and the correction factor ($fp − $sp) is applied at the final stages of code generation, after the final frame size is known.

2.8 The control flow graph

Control flow in *DCode* is created by conditional and unconditional branches, and labels. There are also instructions to call subroutines, and an indexed jump which is dispatched through a *jump table*.

2.8.1 Testing and branching

There are a number of instructions which leave a Boolean result on the top of the abstract stack. These instruction exist for signed and unsigned integers, and for the two precisions of floating point values. As well, there is an instruction `setIn` which tests the value of a single bit in a word on the abstract stack, and two instructions which test set inclusion.

These instructions form the basis for most conditional branching. In general the *left-hand* comparand is pushed on the stack; the *right-hand* comparand is pushed on top, and the particular test invoked. The relational instruction pops its two operands, and leaves either a zero or one value on the top of the stack.

The two instructions *brTrue, brFalse* branch to the given label if the top of stack value is one or zero respectively. The Boolean is popped by the conditional branch.

To be precise, conditional branches are taken depending on whether the top of the abstract stack is zero or non-zero, rather than zero and exactly one. This allows a useful optimization to be performed for comparisons with zero, and reflects a capability which is present in most target machine architectures.

```
; unoptimized IF local = 0 THEN
    pshFP -4
    derefW
    pshZ
    relEQ
    brFalse elseLabel
    ...
```

```
; optimized IF local = 0 THEN
    pshFP -4
    derefW
    brTrue elseLabel
    ...
```

2.8.2 Procedure calls

Procedure calls are either explicit, in the case that the target procedure is known at compile time, or via an address expression which is computed to the top of the stack. The computed procedure call is used for Modula's procedure variables, and for language *C*'s calls of pointers to functions.

call statically determined procedure ...
`call` *procedureName,N*
...
call dynamically determined procedure ...
push value of procedure variable
`popCall`,N

The `popCall` instruction, as its name suggests, pops the procedure entry address from the top of stack.

In the case of object oriented languages with polymorphic (dynamic) method dispatch, the process for computing a procedure address to the top of the abstract stack may be quite complex.

2.8.3 Jump table implementation

The syntax of *DCode* allows for jump tables to be declared at the end of procedure bodies, and provides instructions to perform indirect jumps indexed on these tables. There is no restriction in the way that these constructs are used, except that the jump table entries must be local *label*s. If it should be necessary to produce a jump table that has entries which are symbols known to the linker, then it is necessary to declare the table as constant data with *.ADRS* elements. This is the recommended technique for the construction of method dispatch tables for object oriented languages. However, the use of jump tables declared as constant data to jump *within* procedures is deprecated, as it will generally make it impossible for backends to construct the *control flow graph* of the procedure.

Switch and case statements may be implemented by means of jump tables. The number of entries in the jumptable will be equal to the range of non-default ordinal values in the case labels. Here is a simple example from Modula —

```
   CASE value OF
   | 3,7 : x := 0;
   | 5   : x := 1;
   ELSE ABORT;
   END;
END Proc;
```

Figure 2.5 is the corresponding section of code in the output, for the case where the configuration file has declared the size of the jump-table element as 4.

Note that the code which indexes into the table precedes all code for the cases, preserving the forward control flow property which is useful in some register allocation algorithms. The code for the default case is emitted last in this example, but this is not a requirement.

It would be possible for a backend to optimize low-density jump tables by transforming them into sparse table structures. However, an optimization of this kind is probably better done in the frontend.

```
        pshAdr label2      ; address of table
        pshAdr _value      ; get selector
        derefW
        pshLit 3           ; subtract min value
        sub noTrap
        dup1               ; do bounds check
        pshLit 5
        crdGE
        brTrue label3      ; jump to default
        pshLit 2           ; multiply by 4
        shLeft
        addAdr             ; index into jumpTab
        derefW             ; fetch target address
        jump               ; jump to target
label4:
        pshZ               ; first case
        pshAdr _x
        assignW
        branch label1      ; jump to end
label5:
        pshLit 1           ; second case
        pshAdr _x
        assignW
        branch label1      ; jump to end
label6:
label3:
        pshAdr _value      ; load up value for
        derefW             ; runtime system message
        mkPar  4.0
        trap _gp_casTrp,1  ; default
.ENDCASE
label1:
        exit
        endF
.JUMPTAB label2:
        label4,label6,label5,label6
        label4             ; label6 is the default
.ENDP
```

Figure 2.5: Encoding of simple case statement.

2.8.4 The forward control flow property

Existing *DCode* backends rely on the the fact that, except for loops, branches are always *forward* in the code. This property is easy enough to guarantee for languages which do not have goto statements, and should be attempted even for languages which do have gotos.

The purpose of this restriction is to ensure that for simple interpretive code generators the defining occurrence which begins every live range occurs in the output stream before the last used occurrence of the value. This allows live ranges to be computed in a single pass over a code buffer. The property is actually not required if frontends can guarantee that the abstract stack is empty at all branches, and that no frontend temporaries are live across a branch. However, it is generally too restrictive to demand these latter two conditions, and the forward control flow property is less onerous.

An exception to the forward jump rule occurs for loop constructs, where the back-edge of the loop violates the property. It is easy for code generators to cope with such exceptions however, provided the occurrences are clearly marked in the *DCode*. Consider the code in figure 2.6.

```
       WITH exp DO
         LOOP
            ...
            last use of exp
            ...
         END;
       END;

; DCode resulting from loop inside with
       <compute exp>
       mkTmp   8        ; temp has allocated location
       pop1             ; discard the temp from stack
.LOOP loopLabel:
       ...
       pshTmp  8        ; last use of the temporary
       ...
       branch loopLabel ; the loop back-edge ...
.ENDLOOP
```

Figure 2.6: Skeleton of loop in *DCode*.

All code generators will attempt to allocate a machine register for the with-statement temporary, even though it has had a stack frame location pre-allocated for it. The problem is that the temporary is live across the back-edge of the loop. A naive estimate of the live range of the temporary might believe that it ends at the marked point, in the interior of the loop. Clearly, if this value is allocated to a register, and the same register is reallocated in the second half of the loop a disaster will have occurred. Note that it is not any safer to remove the temporary, and access the value from the abstract stack — the value may still end up in a register, the live range of which will be underestimated by a naive algorithm. In fact, it is a simple matter to avoid the problem when the loop start and end are clearly identified. Any value ranges which are live at a loop entry label have their live ranges arbitrarily extended as far as the end of the closest enclosing loop.

For languages with gotos, it is suggested that the frontend detect backward jumps in the control flow graph, and bracket them with dummy loop begin and end markers. Even if the construct is not actually a loop, this subterfuge will ensure that live ranges are not erroneously truncated. It would be possible for backends which perform global control flow analysis to detect the back-edges themselves, but requiring this is not part of *DCode*.

2.8.5 Stack values live across labels

Existing backends do not insist that the abstract stack is empty at labels in the *DCode*. However, it is necessary to insist that the stack has precisely the same contents along all edges leading to a label, including the "fall-through" path (even if the fall-through path is never exercised). This allows interpretive code generators to ignore flow of control instructions in simulating the abstract stack.

Not insisting that the stack be empty makes it simpler to implement such things as inline range tests. The skeleton of a test which ensures that a value is in the range [-5 .. 5] is shown[3] in figure 2.7. In this example, note the use of an old 2's complement trick: performing a

```
       <compute value to be tested>
       dup1           ; test will destroy one copy
       pshLit 5       ; add lower limit to value
       add            ;
       pshLit 10      ; push (high-low)
       crdLE          ; unsigned compare!
       brTrue okLabel ;
       <get parameters of trap call>
       trap rangeError,3
okLabel:
       <now use the top of stack value>
```

Figure 2.7: Typical range test in *DCode*.

double ended test with a single unsigned comparison. Note also that the abstract stack has the tested value on it during the whole of the testing and branching involved in the range test. There is no problem here, since the stack is unchanged by the flow of control in the range test.

Much more problematic is the example in figure 2.8. Which arises from code such as —

```
boolVariable := expression1 AND expression2;
```

In the erroneous *DCode*, the code attempts to compute a different value along each branch of the control flow, leaving a different final value depending on the actual path. Interpretive

[3] Usually such range tests would be abstracted away with a `test` instruction, but frontends have the option of producing the tests in the way shown in the example here.

```
; BEWARE, THIS EXAMPLE IS ERRONEOUS ...
trueLabel:
     pshLit 1            ; push true Boolean and jump
     branch   mergeLabel
falseLabel:
     pshZ                ; push false and fall through
mergeLabel:
     pshAdr boolVariable
     assignB             ; store the value
```

Figure 2.8: Erroneous attempt to assign a Boolean value.

code generators will erroneously believe that they have a literal on the top of stack when label *mergeLabel* is reached. Clearly the value isn't a literal, it is an anonymous temporary variable, which will take values from the set {0,1} at runtime.

A correct way to handle these occurrences is to move the value off-stack. This may be done by creating a memory location for the value as shown in figure 2.9. An even better[4] solution for the particular example given here, would be to duplicate the assignment along each branch! In effect, the Boolean assignment in the example has been translated into a

```
; THIS IS A CORRECT VERSION ...
     ...
trueLabel:
     pshLit 1            ; push true Boolean
     pshAdr boolVariable
     assignB             ; assign the true value
     branch   mergeLabel ; stack is now empty
falseLabel:
     pshZ                ; push false Boolean
     pshAdr boolVariable ; assign the false value
     assignB             ; stack is now empty
mergeLabel:
```

Figure 2.9: Correct attempt to assign a Boolean value.

conditional expression. Such expressions occur in more obvious form in certain *Algol*-family languages. Language *C*'s conditional expressions are a familiar instance. Figure 2.10 is an

[4]The plausible idea of using the function return register as an off-stack location in which to merge the values is defeated by machine architectures with register windows. In such architectures, the register used by `popRet` and the register used by `pshRet` have different names!

example which shows the correct translation of the expression —

$$(\mathrm{exp}_1 ?\ \mathrm{exp}_2 : \mathrm{exp}_3)$$

The frontend must allocate a temporary (in the example at (\$fp − 24) in the local stack frame), and assign to this location along each path. The resulting value is pushed on the stack after the flow-of-control merge point.

```
        <compute exp1 onto the stack>
        brFalse  fLabel                 ; jump if not zero
        <compute exp2 onto the stack>
        pshFP    -24                    ; $fp-24 is temp address
        assignW                         ; assign exp2 to temporary
        branch   xLabel
fLabel:
        <compute exp3 onto the stack>
        pshFP    -24                    ; $fp-24 is temp address
        assignW                         ; assign exp3 to temporary
xLabel:
        pshFP    -24
        derefW                          ; value is on top of stack
```

Figure 2.10: Push the value `IF exp1 <> 0 THEN exp2 ELSE exp3 END`.

2.9 Use of temporaries

In this section two separate mechanisms for handling off-stack locations are described. The first, the use of `mkTmp, pshTmp` is well established in existing frontends. Because of the existing semantics, the mechanism is unsuitable for use as the primitive to support merging of values. Thus a different pair of instructions `pshFi, popFi` is used for this purpose.

2.9.1 mkTmp and pshTmp

The *DCode* emitted by **gp2d** uses the pair of instructions `mkTmp, pshTmp` to encapsulate the idea of anonymous temporary values. These temporaries are used by **gp2d** for such things as the base address specified in the *WITH* statement, and the upper limit expressions in *FOR* loops. *DCode* preprocessors which (for example) perform common subexpression elimination may allocate new temporaries using the same mechanism.

These temporaries are allocated space in the fixed part of the stack frame, with the argument to the instruction specifying the offset. Note carefully that in the *stackInverted* case the actual offset in the stack frame is the negative of the specified offset. Native code backends will strive to move these temporaries to registers, so that the values will only ever be written to the allocated locations in the event that register spilling becomes necessary.

The `mkTmp` instruction causes a non-destructive store of the value on the top of the abstract stack. It does not pop the value, so no `dup1` instruction is necessary prior to the store. `pshTmp` pushes the saved value onto the top of the abstract stack.

It is important to know that frontends may allocate temporaries for known live ranges corresponding to the structured statements of the source program being compiled. Temporary locations in the stack frame may be reused. Therefore, in the backend there will not be a one-to-one correspondence between the frame offsets of temporaries and the registers allocated to them.

Register allocation and spilling mechanisms are traditionally of two extreme kinds. There are those which assume that all values are "born in memory" and will be moved into registers if a register can be made available. On the other hand, values may be "born in registers" and spilled to memory if the supply of registers becomes exhausted. The two main approaches to register allocation by graph coloring, those of Chow and Hennessy[2], and Chaitin[3], respectively use these two different approaches.

The model used here leads naturally to a register allocation mechanism which is midway between the two extremes. Simple interpretive code generators based on *DCode* will allocate registers to transient values on the abstract stack. They will also allocate registers to the off-stack temporaries introduced by `mkTmp` instructions, if the register pressure is not too great. With most modern architectures it is almost impossible to run out of registers for expression evaluation alone. Thus, in the event that registers require spilling, it is almost always the case that registers containing memory temporaries can be spilled into the pre-allocated locations. This considerably simplifies register spilling.

A logically different cause of register spilling can occur in the case that register values are live across procedure calls. It is usual to have the register set partitioned between *caller-saves* and *callee-saves* registers. Caller-saves registers which are live across procedure calls must be saved and restored around the call. Callee-saves registers which are modified by the procedure must be saved and restored as part of the procedure prolog and epilog. It is the responsibility of the backend to allocate memory temporaries for these purposes, and to expand the stack frame to take account of any such spilling.

Tools to perform *DCode* to *DCode* transformations which involve the invention of new temporaries should allocate these by expanding the stack frame between the fixed-size copies and the open array copies. Allocation of new temporaries thus starts from an offset given by the *.SIZE* value in the procedure header. With this choice, the offsets of all objects allocated by the frontend will remain unchanged. All such tools should be careful to note that temporaries which are of type *double* should allocate 8 bytes, and respect the alignment constraints of the target architecture.

It is dangerous to assume that backends will perform sufficient dataflow analysis to know when separate definitions of temporaries which share the same memory address need to be allocated to the same machine register. Thus if temporary values need to be merged at some control flow point, the values should be *assigned* to explicit stack locations, rather than using the `mkTmp, pshTmp` pair. Back-ends which perform global common sub-expression elimination will move the temporary to a register anyway, and simpler backends will still produce correct code. Thus an attempt to solve the "problem" demonstrated in figure 2.8 by using two `mkTmp`s and a `pshTmp` would still be erroneous.

2.9.2 popFi and pshFi

These instructions were added to support those frontends which create *static single assignment* form (SSA) in the intermediate code. One of the references[4] is the standard work on transformation into this form.

The two instructions provide for the merging of values along different control paths which is required by this form. However, the use of this construct to merge values is now deprecated. When values need to be merged along different control paths, it is preferred that the frontend allocate a dummy local variable, with '(0,0,0)' attributes (see section 1.4.1), and merge values by assigning to this variable. This particular attribute assignment will ensure that the variable is considered for allocation to a machine register.

It is probable that frontends will make little use of these instructions, but they might be used by optimizers which perform *DCode* to *DCode* transformations.

2.10 Test and trap instructions

2.10.1 Front-end, backend and runtime system

It is a design goal that as far as possible frontends should be target independent, and backends should be language independent. With *DCode* this goal is approached by parameterizing frontends for those target attributes which are unavoidable. Back-ends should be entirely language independent, at least for conventional procedural languages.

Both frontends and backends require knowledge of the facilities of the runtime support system. This poses a challenge, since most languages require language-specific runtime support, particularly if they attempt to provide error messages which are presented in terms of the source language semantics. Unless care is taken, knowledge of language specific runtime support diffuses into backends, defeating the objective of language independence.

In the **gardens point** compilers this challenge is met by separating the runtime support for any particular language into two parts. One part is language independent, and includes such things as range and index bounds checking, and the trap handler for hardware traps. The other part is language dependent, and contains such support procedures and traps as are specific to the particular source language.

Back-ends are permitted to be dependent on the facilities of the generic runtime support system, but should never have knowledge of the language specific facilities. This implies that whenever a frontend requires object code to reference a particular facility of the language dependent part of the runtime support, the *name* of the facility must be passed as an explicit name in the *DCode*.

2.10.2 The test instruction

The `test` instruction provides an abstraction for integer bounds-check and trap operations. The semantics of the statement call for the performing of the test, with the specified trap being called if the test fails. All tests are performed on the datum on the top of the abstract stack. The stack is popped following the test. The test may be either expanded by the backend (probably after local optimizations), or directly implemented by machine instructions where available. It is permissible for the backend and the generic part of the runtime system to agree on how diagnostic information is passed to the trap handler.

The form of the instruction is —

`test` *trapName, lowBound, highBound*

where *trapName* it the name of the runtime system trap which should be called if the test fails, while the integer parameters *lowBound, highBound* are the lower and upper bound values respectively.

The `test` instruction is translated in essentially two ways. Machines which have bounds-test instructions may simply invent a new label for the bounds data block, and use this directly in the encoding. Figure 2.11 is an example, for the Intel 386 architecture, of a test instruction and the resulting code (the assembler format is USL SVR4). In this case the generic part

```
<value is on top of stack>              <load value into edx (say)>
test __gp_rTrpLHI, -5,+7                bound  .L003,%edx
...                                     ...


                                        .data
                                  .L003:
                                        .long   -5     /lower bound
                                        .long   7      /upper bound
                                        .long   _gp_rTrpLHI
```

Figure 2.11: Encoding the test instruction for Intel-386.

of the runtime system must have a trap handler which catches the trap, follows the saved instruction pointer to find the bounds-data address, the pointer to the actual trap location and the tested register contents, and then calls the specified trap.

For machines which do not have a bounds test instruction, it is still preferable to make the loading of parameters to the trap-call out-of-line. Then, innocent code does not have to lose time with a taken branch over the trap call.

Figure 2.12 shows how the same example would be encoded for the *MIPS* architecture. For this architecture, the runtime system is called directly, without any intervention by the hardware signalling mechanisms. Note that in this case, as before, the extended basic block containing the test is not broken by a label and branch.

2.10.3 The trap directive

The *.TRAP* directive does for assertion tests what the out-of-line implementations of the `test` instruction does for bounds checks. The syntax of the directive is —

```
<value is on top of stack>          <load value into $r3 (say)>
test __gp_rTrpLHI, -5,+7            addu $r1,$r3,-5
                                    bgtu $r1,12,$L03
...                                 ...


                               # out of line trap calls in code seg
                               $L03:
                                    li   $r4,-5       # load params
                                    li   $r5,7        # to trap call
                                    move $r6,$r3
                                    jal  _gp_rTrpLHI  # call it
```

Figure 2.12: Encoding the test instruction for *MIPS*.

.TRAP *trapName, labelName,* other args ...

The first argument is the name of the trap. The second identifier is the name of the dispatch label, which is the branch destination used in the *DCode* when the trap is invoked. The other arguments, in order, are used as arguments for the resulting trap call.

Figure 2.3 contained an example of use of this feature, and another example is in Figure 2.13. The backend will treat the trap as notification that an out-of-line label *labelX* must be

```
.TRAP __gp_assTrp,labelX,__mNam,25  ; declare the trap label
<evaluate cond1 on stack>           ; Assert(cond1 ...
brFalse   labelX
<evaluate cond2 on stack>           ;          ... AND cond2);
brFalse   labelX
...                                 ; fall through if ok
```

Figure 2.13: Encoding a complex assert test.

emitted in the code segment, followed by a sequence of instructions which load the message parameters, and then call the trap (in this case to *_gp_assTrp*).

It would have been possible to use another variant of the `test` instruction to take assertion traps out of line, but there are compelling reasons not to do so. In order to use a bounds test it would be necessary to evaluate the Boolean expression as a value on the top of the abstract stack. For any but the simplest expressions it is better to never evaluate the Boolean, but to pre-assign a label as the failure target, and to let the the code generator emit possibly multiple branches to this label. The figure 2.13 is a simple example of this, in which the code generator has emitted two branches to the assert trap, arising from the assertion

```
ProgArgs.Assert(cond1 AND cond2);
```

2.10.4 Debugger support

Providing support for symbolic debuggers poses some challenges for compilers which split processing between separate frontends and backends. Some information, such as the structure and name of the types of variables, is available to the frontend. Other information, such as whether a local variable is in memory or register, is known only to the backend after register allocation has taken place.

At this stage, *DCode* provides minimal facilities for the passing of information between backend and frontend. The pragma facility, described in section 2.3.1, allows arbitrary lines to be passed directly from the frontend to the backend output file. Thus the declaration of types, variables and other symbolic entities may be performed by the frontend *provided that the frontend understands the format of the information as it must appear in the assembly language output file.*

It is also assumed that only the frontend will know the relationship between named entities which appear in the object code and the source language names which a programmer would expect when debugging a program. Thus the pragma facility may be used to pass information to the debugger about the *unmunged* or *demangled* names of objects.

In the case of local variables, the frontend knows the type of the variable, but cannot know the storage class. In this case, the ability to place an arbitrary literal string after a local declaration allows the frontend and backend's information to be merged to produce a suitable symbol table entry. If the literal string contains all the type information which is needed by the debugger, then the backend need not interpret the information, but may include the string verbatim in the symbol table entry.

Note that this division of responsibility requires backends to understand the syntax of symbol table entries, but does not require that they understand the names of types, the name demangling strategy or the semantics of the type construction facilities of the source language. Similarly, frontends must understand the format of type declaration symbol table entries.

The symbol table entries for line numbers are entirely generated by the backends, so that the different requirements for (say) *COFF* and *ELF* object formats need not be known to the frontend.

2.11 Problem areas

DCode was intended to be an environment capable of being used for any procedural language. There are a small number of problem areas with certain constructs of common languages. These are detailed in this section, and possible approaches to solutions suggested.

2.11.1 Static links for uplevel addressing

Existing backends, since the time of version one of this document, have relied on the use of a display vector mechanism for non-local addressing. The alternative *static link* mechanism is more suitable for languages such as Pascal, which in some dialects allows nested procedures to be passed as parameters. For implementations of such languages the parametric procedure value is a closure, and is most simply represented as a two element structure comprising the procedure entry point, and the static link.

When a non-local variable is accessed via a static link, the code uses the *difference* in lexical level between the current procedure and the accessed frame to determine the number of dereferences required. This can only work if *every* procedure reveals its lexical level, a necessity specifically rejected in this document. *DCode* only requires those procedures which have uplevel addressed objects to declare the display vector level. This is a simple optimization, since if a procedure has no uplevel accessed data there is no need to waste time saving and restoring the current display location.

DCode could be changed to force procedures to declare their lexical level, *as well as* stating the need for a display vector save/restore operation. But in the meanwhile, explicit encoding in the *DCode* is simple and effective.

One solution – explicit code

Static links may be encoded by simply giving all procedures the *.NODISPLAY* attribute, and passing the static link as an additional parameter to the nested procedure. This involves allocating a local memory location for the static link of each nested procedure. Since the lexical ancestor of a called procedure is always either the caller or some known ancestor of the caller, it is possible to compute the static link in the way shown in figure 2.14. In this example it is assumed that parameters are passed on the stack, and that the static link is pushed as an additional parameter immediately before the call. The static link will thus be at ($fp + *stackMarkSize*) of each procedure. The example shows two calls. In one of these the calling procedure is the ancestor, and so the stack frame pointer of the caller is the static link. For the second call the stack frame pointer of a common ancestor *Level1* is passed as static link to a call to the procedure *Level2*. It is assumed that the stack mark size is 8.

In order to access non-local data it is necessary to traverse as many static links as the difference in lexical level between the accessor and the accessee. Figure 2.15 gives the code which would be used by the procedure *Level4* of the example in figure 2.14 to access a datum in procedure *Level1*, with a difference in lexical level of three.

2.11.2 Environments requiring "Pascal call conventions"

Some environments use conventions in which the order in which parameters are pushed onto the runtime stack is opposite to that customarily used for language *C*, and which require the called procedure to remove the parameters from the stack before returning to the caller. These conventions are usually called *Pascal call conventions*.

Firstly, it should be noted that the relationship between the order in which parameters appear in the source code and the order in which they are placed in the runtime stack frame

```
.PROC Level3( ... ,.NODISPLAY)
          ...
; code to call a nested procedure "Level4"
      <push params for Level4>
      pshFP  0          ; $fp is the static link
      mkPar  4          ; push as dummy final parameter
      call   Level4,1
      ...
; code to call an uplevel nested procedure "Level2"
      <push params for Level2>
      pshFP  8          ; get static link of Level3
      derefW            ; which is $fp of Level2
      addOff 8          ; get static link of Level2
      derefW            ; which is $fp of Level1
      mkPar  4          ; push as dummy final parameter
      call  Level2,1
      ...
```

Figure 2.14: Code for passing a static link to a called procedure.

```
.PROC Level4( ... ,.NODISPLAY)
          ...
; code to access frame of "Level1"
      pshFP   8         ; get static link of this proc
      derefW            ; this is $fp for Level3
      addOff  8         ; get static link of Level3
      derefW            ; this is $fp for Level2
      addOff  8         ; get static link of Level2
      derefW            ; this is $fp for Level1
      addOff  -24       ; at last, the address
      derefW            ; and finally the datum
      ...
```

Figure 2.15: Code for accessing a datum at offset -24 with lexical difference 3.

is transparent to *DCode*. Thus the order of passing parameters is not an issue. Front-ends may map source code parameters to the runtime stack frame in any order which is convenient.

Secondly, it should be noted that in the case of machines which use a fixed parameter-assembly area neither caller or callee remove parameters from the runtime stack. Thus in this case also the "call conventions" have no impact on the detail of the *DCode* interface.

Finally, in the case of compiled procedures which make (for example) system calls to an application programming interface using Pascal conventions there is no problem. It is only necessary for the frontend to know that certain calls possess those semantics, and to refrain from emitting the `cutPars` *number* instruction which would otherwise be placed immediately after the `call` instruction in those cases. Thus it is simple for procedures defined in *DCode* to *call* foreign procedures with Pascal call conventions.

If for some language it is necessary to *define* procedures in *DCode* which cut parameters from their stack frames prior to returning to their caller, then the *.RETCUT* declaration must be included in the procedure header.

Appendix A

The runtime environment for gp2d

A.1 Facilities of the runtime system

Figure A.1 details the facilities provided by the generic runtime support library of the **gardens point** compilers. There are additional facilities required for each language, beyond the generic facilities here.

The runtime system also contains the signal handlers which catch hardware signals, and produce the appropriate abort messages. These are never called directly, and hence are not exported.

A.2 The configuration file format

This section indicates the typical information which is required for a frontend language processor to produce *DCode* which is parameterised for particular targets. The information in this section is thus not part of the *DCode* specification, but simply indicates a possible solution to the parameterization problem.

The configuration file allows the frontend to be configured for a variety of different machine conventions. However, all machines are assumed to use the *IEEE* format for real numbers and 2's complement for the integer types. The format is shown in the Table.

The characteristics which may be varied are, the parameter passing conventions, the direction of stack growth, the object alignment properties, and the byte packing order (*endian*-ness) of the underlying machine.

The meaning of the various commands is —

-Endian this declares the target machine as being either *littleEndian* (least significant byte has lowest address) or *bigEndian* (most significant byte has the lowest address).

stack- this declares the runtime stack of the target machine as growing toward lower addresses (*Inverted*) or higher addresses (*Upright*). The inverted case is more common.

```
INTERFACE DEFINITION MODULE gprts;
  IMPORT SYSTEM;

  (* pointer to the current coroutine vector, the first 16  *)
  (* words of which are the display vector for the thread   *)
  VAR _currentCo : POINTER TO ARRAY [0 ..15] OF SYSTEM.ADDRESS;

  VAR _stackLim  : SYSTEM.ADDRESS; (* limit for stack trap  *)

  PROCEDURE _gp_timTrp() : INTEGER;(* retrn secs since 1970 *)
  PROCEDURE _gp_stackTrp(); (* abort with stackover message *)
  PROCEDURE _gp_funTrp();   (* abort with bad func message  *)
  PROCEDURE _gp_storTrp();  (* abort with storage error     *)

  (* abort with bad case selector ORD = badSelector message *)
  PROCEDURE _gp_casTrp(badSelector : INTEGER);

  (* abort with assert message ---                          *)
  (*   if lineNo is zero just print message, else message   *)
  (*   is the module name and lineNo is the aborting line   *)
  PROCEDURE _gp_assTrp(message : ARRAY OF CHAR; (* no HIGH  *)
                       lineNo  : CARDINAL);

  (* abort with various range trap messages,                *)
  PROCEDURE _gp_rTrpLHI(min, max, val : INTEGER);
  PROCEDURE _gp_rTrpLHU(min, max, val : CARDINAL);

  (* abort with various index trap messages,                *)
  PROCEDURE _gp_iTrpLHI(min, max, val : INTEGER);
  PROCEDURE _gp_iTrpLHU(min, max, val : CARDINAL);

END
```

Figure A.1: Interface to generic runtime system

-Params this declares the parameter assembly strategy as being to push parameters on the stack incrementally (*stackParams*), or to use a fixed size stack frame with a parameter assembly area in the top of each frame (*fixedParams*). If parameters are to be passed in registers then *fixedParams* must be declared. It may be advantageous in other cases also.

stackMarkSize this declares the number of bytes occupied by the stack mark, including return address and dynamic link, but not the display save location.

displayElSize this declares the size of the display element, except for a few unusual cases, this should be word size.

file	→	{command ';' } eofSy.
command	→	'littleEndian' | 'bigEndian'
	|	'stackUpright' | 'stackInverted'
	|	'fixedParams' | 'stackParams'
	|	'stackMarkSize' '=' *number*
	|	'displayElSize' '=' *number*
	|	'jumpTabElSize' '=' *number*
	|	'maxFieldAlign' '=' *number*
	|	'maxParamAlign' '=' *number*
	|	'minParamAlign' '=' *number*
	|	'inlineTests'
	|	'externRecRetPtr'
	|	'externArrRetPtr'
	|	'wordSize64'
	|	'refRecords'
	|	'sunValueStructs'
	|	'gdbStabs'
	|	*ident* '=' 'calleeCutPars'

Table A.1: The extended bnf for the configuration file

jumpTabElSize this declares the size of the jump table elements, except for a few unusual cases, this should be word size.

maxFieldAlign this declares the most stringent alignment constraint of the target machine. All primitive objects less than or equal to this size will be aligned on a boundary equal to their size.

minParamAlign this declares the smallest parameter size which is permitted. For example, the iapx86 permits byte alignment for all objects, but requires parameters to be even aligned as *push* always pushes a pair of bytes.

maxParamAlign this declares the most stringent alignment for parameters. By default this is the same as *maxFieldAlign* but there are deviant architectures such as *SPARC* which octo-align floating point double record fields, but only quad-align double parameters.

inlineTests this declares that the target backend is incapable of handling the `test` instruction. In such a case the tests must be expanded inline in the code, by means of explicit comparisons. It is always faster to use out-of-line tests if the backend supports these, since in that case error-free code does not have a taken branch over the trap call.

refRecords this declares that value records are passed by reference and will be copied by the callee. The default is to pass value records by value, and value arrays by reference.

sunValueStructs this declares that value records are passed by reference, with the copy being made by the *caller* rather than the *called procedure*. This flag requires that *refRecords* be already declared in the configuration file.

externRecRetPtr this declares that functions returning record-valued results expect their result return pointers to be in the architecture defined dedicated location, rather than passed as a dummy first parameter.

externArrRetPtr this declares that functions returning array-valued results expect their result return pointers to be in the architecture defined dedicated location, rather than passed as a dummy first parameter.

wordSize64 this declares the target machine to have 64-bit words, rather than the 32-bits assumed as default.

calleeCutPars this declares the identifier as a context sensitive mark which parsers recognize as tagging modules with Pascal call conventions. This allows for system-specific configuration files to specify meaningful system-specific marker names.

gdbStabs this declares that the target assembler can understand symbol table entries in the format used by the Free Software Foundation's debugger `gdb`. If this flag is present and the **−g** command line switch is used, then the *DCode* output will have embedded debugging pragmas as described in 2.10.4.

The format of the file is free, and Modula comments may be included. It is scanned by the standard scanner, and thus would even allow numbers to be given in octal or hexadecimal format.

Example configuration files for iapx86, Hewlett Packard HP-PA and SUN microsystems *SPARC* architectures are as follows–

Configuration file for iapx86

```
(* cfg for iapx86, 16-bit segments, large memory model *)
stackMarkSize = 6;
displayElSize = 2; (* only offset is needed *)
jumpTabElSize = 2; (* only offset is needed *)
maxFieldAlign = 2; (* faster than 1 on new micros *)
minParamAlign = 2;
stackInverted;
littleEndian;
stackParams;
```

The above example is for iapx86 with 16-bit segmented memory model. The 32-bit flat address space used by Intel 80386 under OS2 V2 and V3, Windows NT and UNIX would be —

```
(* cfg for 32-bit flat address space iap386,486 etc. *)
stackMarkSize = 8;
displayElSize = 4;
jumpTabElSize = 4;
maxFieldAlign = 4;
minParamAlign = 4;
stackInverted;
littleEndian;
fixedParams;
gdbStabs;        (* for Linux version *)
```

The above example shows two innovations. Firstly, even for machines which pass their parameters on the stack there is some advantage in using fixed assembly, and this is done on current versions of **gpm** for the intel architecture. Also, for versions which use assemblers which understand `stabs`, and use `gdb` for debugging, the declaration of `gdbStabs` causes debugger information to be placed in the *DCode* output.

Configuration file for HP-PA

```
(* cfg file for HP precision architecture *)
stackMarkSize = 8;
displayElSize = 4;
jumpTabElSize = 4;
maxFieldAlign = 8; (* fp dbl on octo-byte *)
minParamAlign = 4;
stackUpright;      (* this is very unusual *)
bigEndian;
fixedParams; (* must use fixed, for alignment *)
```

Configuration file for SPARC

```
(* cfg file for SPARC, to match C conventions *)
stackMarkSize = 68; (* mark + window save *)
displayElSize = 4;
jumpTabElSize = 4;
maxFieldAlign = 8; (* fp dbl on octo-byte *)
minParamAlign = 4;
maxParamAlign = 4; (* dbl pars only quad! *)
stackInverted;
bigEndian;
fixedParams;
externRecRetPtr;   (* functions which return C struct
                      values have the pointer to the
                      destination location passed
                      in a special, dedicated location *)
refRecords; sunValueStructs;
gdbStabs;          (* produce information for gdb      *)
```

Configuration file for Alpha-AXP

```
(* cfg file for Digital Alpha *)
stackMarkSize = 0;
displayElSize = 8; (* pointers == 64bits *)
jumpTabElSize = 8;
maxFieldAlign = 8; (* fp dbl on octo-byte *)
minParamAlign = 8;
stackInverted;
littleEndian;
fixedParams;
wordSize64;
```

Appendix B

The gp2d stack frame format

The stack conventions used by **gp2d** are variable to some extent, as defined by the configuration file. However, certain features are fixed. In all cases the order of components of the stack frame are assumed to be as in the figure B.1. This first diagram is an example, for the case of a declaration of *stackParams*, with an inverted stack.

Local variables are the first area in the stack frame but are preceded by the display save area for those procedures which need to save a display location. This moves all other offsets down by *displayElSize*, and must be taken into account by the process which assigns offsets to variables. **gp2d** does this automatically.

The second area in the stack is the temporary area, which holds anonymous variables which are allocated by the front end. It is necessary to have memory temporaries to evaluate large set-valued expressions, and to provide destinations for the return values of functions which return structured values such as arrays or records. The frontend may also use this area as a temporary holding place for common subexpressions. For example, for many *FOR* loops, it is necessary to define a memory temporary to hold the upper bound value, since this must not be evaluated more than once.

The array copy areas come next, with fixed size arrays first and open arrays last. The size of the last area is not known at compile time, so that the declared stack frame size refers to everything below the stack mark in memory down to the limit of the fixed array copy area. On some machines, it may be convenient to keep the stack frame size fixed, and use a separate stack for open array parameters (soap).

In any case, if the backend code generator needs to allocate temporaries this must be done between the top of the fixed array copy area and the start of the open array copy area. Code generators will need to allocate additional temporaries for register spilling, and for the saving of *callee saves* registers. The size of the fixed part of the stack must be adjusted to reflect any additional allocation here.

In the case of machines which require an area for the automatic saving of register windows, the runtime stack frame size will need to be adjusted by the code generator. This is probably best done by incorporating this extra requirement into the abstraction of the "stack mark".

In the case of machines which use a fixed parameter assembly area, the diagram is slightly different

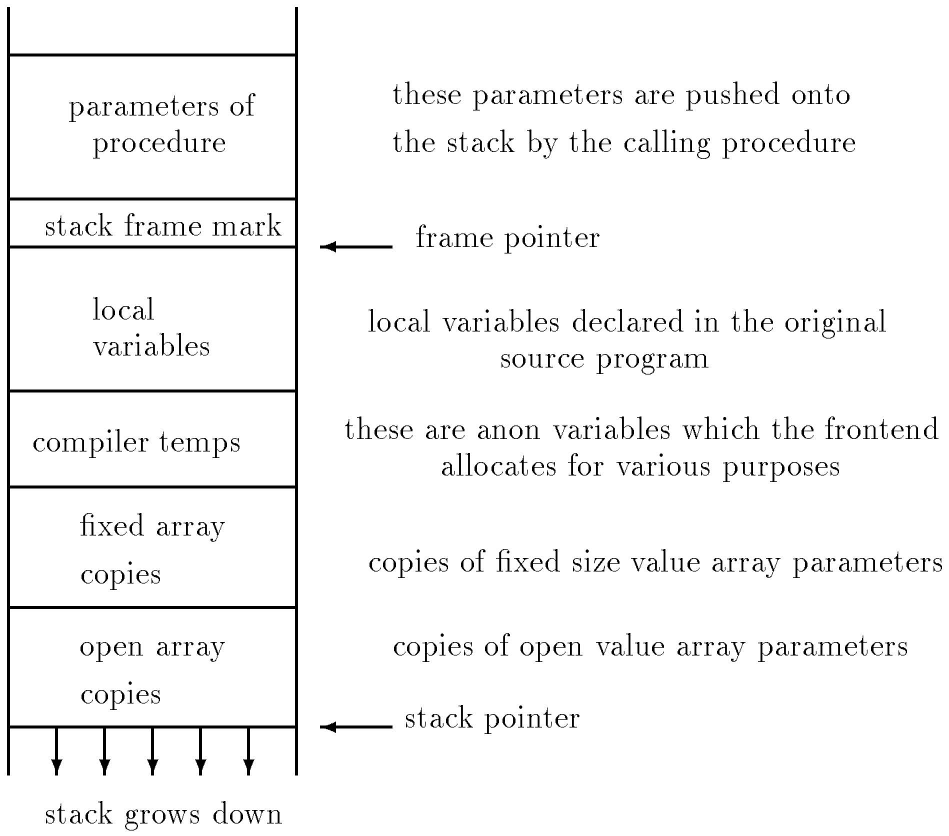

Figure B.1: Stack frame #1: parameters passed by pushing on the stack

B.1 Parameter passing conventions

There are two main strategies for the assembly of parameters, and their passage to the called procedure. The actual strategy is selected by the configuration file.

B.1.1 Parameters pushed on the runtime stack

Parameters are pushed on the stack in reverse order, that is, the first named parameter is pushed last. This means that the first parameter is always at offset ($fp + *stackMarkSize*). Record types, and scalar parameters including floating point types are passed by value if of value mode, and by reference if of variable mode.

Arrays are always passed by reference and value arrays are copied, either by the calling or called procedure. There are significant advantages in having the called procedure make the copy, since for fixed length arrays the procedure may then directly access the copy without the extra indirection. However for languages such as ISO Modula-2, which allows storage size changes during the parameter substitution process, the calling procedure must *marshall* the parameters.

In the case of open arrays the *HIGH* value is pushed first, so that it is at an offset which is one word above the pointer to the actual array. The high value is always word sized. Modula

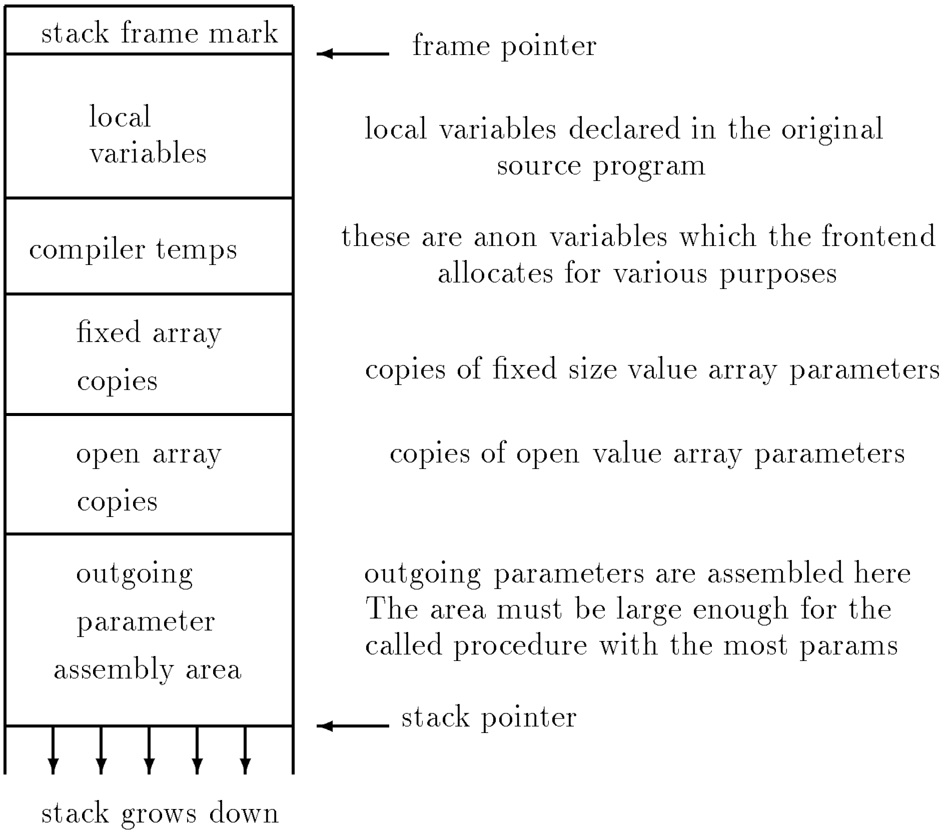

Figure B.2: Stack frame #2: parameters are assembled in a fixed area

sets are treated as scalars if the base type is of cardinality no more than *"bits-per-word"*. Large sets are passed as arrays.

Functions return values off-stack. In the case of scalars this happens automatically through calls to the various *popRet* and *pshRet* instructions. However, in the case of structured values it is necessary to adopt a more complex mechanism. The strategy used by **gp2d** is similar to that used by many *C* compilers. For structured return values the caller defines an off-stack memory temporary as destination, and passes a pointer to this as an additional parameter. The destination pointer is pushed last, after all the parameters. In effect, such functions take an extra variable mode parameter, which appears in the *DCode* as a new first parameter. The function copies the returned expression to the specified destination, but also returns a pointer to the result as a conventional return value.

B.1.2 Parameters assembled in a fixed location

The stack frame has an additional area defined at the top of the frame, which is the parameter assembly area. This area is as large as the parameters of the called procedure of greatest parameter size. In **gp2d** this area is always at least four words, since it is not known if runtime support procedures will be called at the time that the size is evaluated. The most complex runtime procedure requires four words of parameters.

Parameters cannot be evaluated in arbitrary order in the case of fixed assembly. Because the same area is used to assemble parameters for all of the procedures called by any particular procedure, use of the area must be serial. **gp2d** will always assemble the actual parameters in the following order —

- parameters which require function calls are evaluated in some order, but the values are left on the abstract stack until all such parameters have been evaluated
- the evaluated parameters are popped from the abstract stack and moved to the parameter assembly area with successive calls of `mkPar`
- the remainder of the parameters are evaluated in some order, and moved to the assembly area as they are evaluated

The default is to pass arrays by reference, and structures by value, but the configuration file may specify that value structures are passed by reference and copied by the callee.

SUN Microsystems compilers adopt a different strategy for value structures, passing a reference after making a copy in the calling procedure. This has little to recommend it, but is desirable for cross-language calling to *C* libraries.

Bibliography

[1] U Ammann. Code generation for a Pascal compiler. In *Pascal, the Language and its Implementation*, edited by D W Barron, John Wiley and Sons, 1981.

[2] F C Chow and J L Hennessy. The priority-Based Coloring approach to register allocation. *Trans. on Programming Language and Systems.* Vol 12, n4, 1990.

[3] G J Chaitin. Register allocation and spilling via graph coloring. In *Proceedings of the ACM SIGPLAN 82 Symposium on Compiler Construction.* ACM, 1982.

[4] R Cytron, J Ferrante, B K Rosen, M N Wegman and F K Zadeck. Efficiently computing static single assignment form and the control flow graph. *Trans. on Programming Language and Systems.* Vol 13, n4, 1991.

[5] C W Fraser and D R Hanson. A code-generation interface for ANSI-C, *Software Practice and Experience*, Vol 21, Sep. 1991.

[6] J Gough. The portable Modula-2 front end gp2d. QUT FIT technical report 1/92, (also available as University of Tübingen report WSI-92-2).

[7] K J Gough, C Cifuentes, D Corney, J Hynd and P Kolb. An experiment in mixed compilation/interpretation. *Proceedings of ACSC-14*, Australian Computer Society, 1992.

[8] D R Perkins and R L Sites. Machine independent Pascal code optimization. *Proc. SIGPLAN symposium on compiler construction 1979* 201–207, 1979.

[9] R Sethi and J D Ullman. The generation of optimal code for arithmetic expressions *Journal of the ACM* 17:4 715–729 1970.

[10] N Wirth. The design of the Pascal compiler. *Software Practice and Experience* 1:4 309–333, 1971.