Multiflow Computer, Inc.

Intermediate Language Document
Version 4.3

Multiflow Computer, Inc.
175 North Main Street
Branford, CT 06405
(203) 488-6090

July 19, 1994

Copy Number: ______

Revision History

Version 1.0

PGL Extremely rough initial version (IL only).

Version 2.0

PGL Conversion to LaTEX. Addition of FORTRAN section. GADD, etc. were also added.

Version 3.0

TJK Massive changes. Updated all examples. Noted many remaining inadequacies.

Version 4.0

PGL Added chapter on IL-2. Place chapters on C and F77 into separate documents. ⇐4
Updated IL-1. Flagged changes.

Changes to IL-1:

- Allow for overflow trap on byte and halfword stores.
- Added END-RECORD, END-PACKED to avoid syntactic ambiguity.
- Removed base pointers.

- Changed VOTGEN.
- Defined a strict correspondence between index list in array accessing polynomial and the declared dimensions of the array. This is to allow the declared dimensions to be used by an assertion facility in Phase 2.
- Added generalized conditionals (IF-GAND, etc, IF-GBAND, etc.).
- Changed name of SELECT operation to SWITCH.
- Added SLCT-xx operations.

Version 4.1

TJK Many random changes to IL-1 and IL-2. ⇐4.1

Changes to IL-1:

- Removed continuation character, replaced with termination character. Other related syntactic changes.
- Added line numbers and volatile attributes.
- Specified variable names.
- Improved the definitions of constants throughout IL-1. In particular, type information for constants has been fixed.
- Removed CONST-DECL.
- Fixed packet initializations.
- Removed packet-number.
- Fixed incorrect definition of VOTGEN.
- Changed alignment and defined it more precisely.
- Added assertions.
- Added new operators for calling routines with no args.

- Added restrictions on shifts.
- Changed record copy and array copy operations.
- Many other minor changes, corrections, and clarifications.

Changes to IL-2:

- Specified variable names.
- Removed "space-reservation" operand for ENTRY pseudo-ops. It is now part of the symbol table interface.
- Added BACK-EDGE pseudo-op.
- Provided a mechanism for indirect calls.
- Many other minor changes, corrections, and clarifications.

Version 4.2

TJK Updated declaration portion to be conceptually correct. ⇐4.2

- Brought routine attributes up to date.
- Brought packets up to date.
- Clarified packet alignment.
- Clarified packet size.
- Fixed packet allocation.
- Fixed packet loader-info.
- Fixed description of EXTERN and DEFINED packet loader-info.
- Fixed allocation/loader-info table.
- Fixed variable size, esize, and dimension-info.

- Clarified variable arg-info.
- Updated symbol table examples.

Version 4.3

TJK Updated alignment and declaration sections. ⇐4.3

- Cleaned all .tex files.
- Eliminated acceptable alignment.
- Relaxed restriction on number of top-level RECORDs in an ABLOCK packet.
- Changed and clarified alignment restrictions for ABLOCK, PARAM, and TEMPLATE packets.
- Changed and clarified alignment restrictions for RECORDs.

Unresolved issues

In IL-1:

- Restrictions on pointers in GADD and GMUL. (TJK will specify these later.)
- Character string operations. (For now, these will be function calls.)
- Array copy operations. (For now, these aren't supported.)
- Alignment assertions. (For now, these aren't supported.)
- The coercion table. Some of the entries (the ones flagged with question marks) may not be supported by the hardware. If not, they will be removed. (For now, we are leaving them in.)
- IL-1 examples. (For now, these are syntactically incorrect, and many of the alignments are incorrect.)

In IL-2:

- Unspecified integer instructions.
- Floating instructions.
- Moving F64 registers into integer registers. (We will support the hardware.)
- Loading an F32 constant into an F32 register. (We will support the hardware.)
- Calls to routines with no args.

Contents

Chapter 1

IL-1 (high-level subset)

1.1 Extended BNF primer

<table>
<tr><td>::=</td><td>definition</td></tr>
<tr><td>|</td><td>alternate</td></tr>
<tr><td>(a | b)</td><td>a or b</td></tr>
<tr><td>[a]</td><td>0 or 1 a's</td></tr>
<tr><td>{ a }</td><td>0 or more a's</td></tr>
</table>

These symbols must be quoted to appear as terminals.

1.2 Alignment

The concept of *alignment* is used throughout this document. Alignments have the form xMODy where y is a power of 2 (e.g., 1, 2, 4, 8, 16, 32, 64, ...) and x is constrained to be in the range 0..y-1: ⇐4.2

alignment	::=	xMODy

We refer to x as the *offset* and y as the *base* of the alignment. ⇐4.2

Alignments are used to indicate restrictions on the placement of objects in main memory. Alignment xMODy means the address assigned to the first byte of the object must have a remainder of x when divided by y.

0MOD1 is byte alignment, 0MOD2 is 16 bit alignment, 0MOD4 is 32 bit alignment, 0MOD8 is 64 bit alignment. Larger values of y can be used to align values in particular banks. $\Leftarrow$4.2

In examining alignments, we will be interested in alignment compatibility. Alignment x1MODy1 is compatible with x2MODy2 if y2 is a multiple of y1 and x1 = x2 (mod y1). Note that "compatible" is not a commutative relation as defined here. For example, 1MOD4 is compatible with both 1MOD8 and 5MOD8, but not with 2MOD8 or 1MOD2. If A has an alignment compatible $\Leftarrow$4.2
with B, we can force A to be aligned with B while still obeying A's alignment.

The least restrictive alignment is 0MOD1 (arbitrary byte alignment). 0MOD0 is not allowed, since it is meaningless (besides, 0 is not a power of 2).

It should be clear that there is a one-to-one mapping from the alignment values onto the positive, non-zero integers. The integer corresponding to $\Leftarrow$4.2
xMODy is merely x + y. This gives us a convenient enumeration of the possible alignments.

1.3 IL Machine Model

The IL assumes a machine with a load-store architecture. The machine has an unlimited number of named temporaries, which hold values of the primitive types. Simple RISC level operations can be performed on values in temporaries. The machine has an associated byte-addressable memory with a 32 bit address space. The machine can load and store from memory, but can perform no other operations on objects in memory. A conventional sequential flow of control is assumed; parallel semantics will be added at a later date.

The memory-resident scalar data types supported by the IL are listed below, together with their size and their "preferred" memory alignment. The data $\Leftarrow$4.3
types will be represented in memory using the Saturn hardware conventions (i.e., MSB is in byte 0 of a multibyte data type). A scalar is an object of one $\Leftarrow$4
of the primitive data types. An aggregate is a collection of scalars treated as a single object (although the members of bit-fields aren't regarded as normal scalars).

IL type	description	size (bytes)	preferred alignment
I8	8 bit unsigned integer	1	0MOD1
I16	16 bit unsigned integer	2	0MOD2
I32	32 bit unsigned integer	4	0MOD4
SI8	8 bit signed integer	1	0MOD1
SI16	16 bit signed integer	2	0MOD2
SI32	32 bit signed integer	4	0MOD4
F32	32 bit floating point	4	0MOD4
F64	64 bit floating point	8	0MOD8
P32	32 bit pointer	4	0MOD4

⇐4.3

The preferred alignment for a data type will always be used when allocating storage for an object of that type, unless user-specified associations make this impossible. For example, we greatly prefer to have F64 variables aligned ⇐4.3 0MOD8. However, standard Fortran permits situations where F64 variables must be aligned 4MOD8 (due to equivalencing, or the appearance of an F64 at a 4MOD8 position in a common block). Code which accesses misaligned variables will be slow, but hopefully this situation won't arise very often. ⇐4.3

8 bit and 16 bit integers are not fully supported in the IL. They can only be loaded, stored, copied, and converted to other types; the arithmetic and comparison operations are not defined for types of size less than 32 bits. If overflow checking is enabled, we will check for overflow of 8 and 16 bit ⇐4 data types when they are stored to memory. This is what the hardware will support. 8 bit and 16 bit arithmetic is not required by standard C or FORTRAN. However, it is a compatibility issue. It is not included in the IL definition.

prim-data-type	::=	unsigned-integer-type | signed-integer-type | floating-type | pointer-type
unsigned-integer-type	::=	I8 | I16 | I32
signed-integer-type	::=	SI8 | SI16 | SI32
floating-type	::=	F32 | F64
pointer-type	::=	P32

The following data types are not supported.

I64	64 bit unsigned integer
SI64	64 bit signed integer
F128	128 bit floating point

These data types, with the addition of F64, are the primitive data types supported by Cray FTN.

1.4 IL Program Structure

An IL program is a list of operations (including pseudo-operations). Each operation must be explicitly terminated with a semicolon ";". Note that this ⇐4.1 means module-level DCLs, routine-level DCLs, and operations. Variable declarations within a packet are not explicitly terminated with a semicolon, except of course when they are the last thing in a packet.

Whitespace (including newline) is ignored, except within a quoted string. The sharp sign "#" is the comment character. Everything from a sharp ⇐4.1 sign to the end of the line is ignored, unless the sharp sign is embedded in a character constant or character string.

Variable names may be arbitrarily long. ⇐4.1

An operation is a line number, optionally followed by a volatile specifier, ⇐4.1 followed by an operator, followed by its operands. An operator is a generic operator qualified by type and/or size information.

operation	::=	line-number [vol-spec] operator { operand }
vol-specifier	::=	MEMORY-VOLATILE | SUPER-VOLATILE
operator	::=	generic-op { . (op-qualifier) }
op-qualifier	::=	type-qualifier | size-qualifier | other-qualifier | VOID
type-qualifier	::=	temp-data-type | C64 | C128
size-qualifier	::=	32 | 64

Line-number is simply a positive integer indicating the source line which ⇐4.1 produced the operation.

Vol-spec is optional and usually absent. It is provided to enable system code ⇐4.1 to be safely written. The two possible values are as follows:

- MEMORY-VOLATILE. This may only be applied to memory reference operations. It indicates that the referenced memory object may be asynchronously read or written before or after this operation is executed. It imposes two restrictions on the compiler:

  - No MEMORY-VOLATILE operation may leave its original basic block.
  - No MEMORY-VOLATILE operation may pass another MEMORY-VOLATILE operation within a given basic block.

- SUPER-VOLATILE. This may be applied to any operation. It is intended for use with operations which map into special machine instructions (for OS use). It is stronger than MEMORY-VOLATILE in that it includes all MEMORY-VOLATILE restrictions and adds other restrictions. In effect, it turns off most global optimization of any loop containing a SUPER-VOLATILE operation. In particular:

  - No SUPER-VOLATILE operation may leave its original basic block.
  - No SUPER-VOLATILE operation may pass or be passed by any other operation. In other words, in any given execution of the program, no operation may cross the execution of a SUPER-VOLATILE operation, in either direction.

  In practice, each SUPER-VOLATILE operation will constitute its own basic block, and global optimization will not take place.

Other-qualifier can be anything. It is included here for completeness. The more common qualifiers are explicitly listed, but there are a few operators which require special qualifiers with unique interpretations.

Temp-data-type is defined in another section (you didn't really think this was a one-pass document, did you?). It typically indicates the data type of the operands. For example, a signed 32 bit integer divide is written:

```
DIV.SI32 t1 t2 t3
```

t1, t2, and t3 are all temps whose values are interpreted as signed 32 bit integer values, and t1 is assigned the quotient of t2 and t3. Note that the actual types of the temps may not be SI32 (e.g., t2 might be SI8), although for the purposes of the operation they are all treated as SI32 values. It is up to the various phases of the compiler to maintain the validity of type information of temps. For example, t1 may safely be SI8 if t2 is SI8. Temps are described in more detail later.

Size qualifiers specify the number of bits to be operated on in the given context.

If there is only one specific operator corresponding to the generic operator, the op-qualifier is omitted. VOID is used in the CALL and RETURN instructions, to describe the CALL of (RETURN from) a routine which does not return a value.

Pseudo-operations are non-executable, and contain information for the IL translator. They are syntactically similar to operations:

pseudo-operation	::=	psuedo-op { operand }

The basic unit of an IL program is a module, which is a collection of routines which share a common static data area. A module begins with the MODULE pseudo-op, followed by a list of data declarations and a list of routines.

il-module	::=	MODULE module-name { module-level-decl } { routine } END-MODULE
module-level-decl	::=	packet-decl | routine-decl

⇐4.1

A routine is a header, a list of data declarations, a list of operations, and an end.

routine	::=	routine-head { routine-level-decl } { other-oper } END-ROUTINE
routine-level-decl	::=	packet-decl | routine-decl | temp-decl

⇐4.1

Declarations at the module level are global to the module. Declarations in a routine are local to the routine. There is no allowance for nested routines.

1.5 Names and literals

Names are arbitrary length strings of characters. They may contain any of the following:

- The uppercase letters "A"–"Z".
- The lowercase letters "a"–"z".
- The digits "0"–"9".
- The underscore character "_".
- The dollar sign "$".
- The period ".".
- The two-character sequence consisting of a question mark "?" followed by the digit one "1", i.e., the sequence "?1".

The first character in a name must not be a digit "0"–"9" or a period ".". A name is terminated by any character other than those characters which may appear in a name, including whitespace. The appearance of a question mark "?" which is not followed by the digit one "1" therefore does not constitute a break character but rather an illegal name. ⇐4.1

The idea is that each phase of the compiler will be free to introduce names with the sequence "?n", where n is the phase number, without fear of conflicting with a name created before that phase. Of course, at this point in compilation, only Phase 1 will have had a chance to create new names. ⇐4.1

All names in an IL module must be unique. This includes names in different routines, and the names of components and fields in an IL record. The same name may be used in different routines if and only if they refer to the same object. Thus a name in an IL program never needs to be qualified by its context.

There are literal constant representations for all of the primitive data types.

syntactic name	size (bits)	type
I8-constant	8	I8
SI8-constant	8	SI8
I16-constant	16	I16
SI16-constant	16	SI16
I32-constant	32	I32
SI32-constant	32	SI32
P32-constant	32	P32
F32-constant	32	F32
F64-constant	64	F64

⇐4.1

There is no restriction on the bit pattern of any constant, e.g., an I32-constant may be any 32 bit sequence of binary digits. ⇐4.1

It is convenient to group constants by their size:

8-bit-constant	::=	I8-constant
		| SI8-constant
16-bit-constant	::=	I16-constant
		| SI16-constant
32-bit-constant	::=	I32-constant
		| SI32-constant
		| P32-constant
		| F32-constant
64-bit-constant	::=	F64-constant

⇐4.1

It is also convenient to group constants by their uses:

oper-constant	::=	32-bit-constant
		| 64-bit-constant
init-constant	::=	8-bit-constant
		| 16-bit-constant
		| 32-bit-constant
		| 64-bit-constant

⇐4.1

Oper-constants are, in general, the only constants which may appear as operands in IL-1 operations. Init-constants are the only compile-time constants which may appear in initializations. ⇐4.1

The remainder of this section is somewhat out of date. It is left here merely as a suggestion for how we might want to textually represent various types of constants. The details can be worked out by by the front-ends and the IL reader. The important point to keep in mind is that beyond the IL reader, constants will be thought of as bit strings with a length and a type (as indicated in the first table). ⇐4.1

Old ideas:

A signed integer constant is either a decimal constant or a constant in a specified base. A signed decimal constant consists of an optional sign, a sequence of decimal digits, and a type qualifer. A signed based constant consists of a base, a sequence of digits of the appropriate base, and a type qualifier. The type qualifier can be omitted if the type is clear from context.

<table>
<tr><td>signed-integer-constant</td><td>::=</td><td>(([sign] decimal-literal) | based-literal) [‘(’ signed-integer-type ‘)’]</td></tr>
<tr><td>sign</td><td>::=</td><td>+ | -</td></tr>
<tr><td>decimal-literal</td><td>::=</td><td>digit { digit }</td></tr>
<tr><td>digit</td><td>::=</td><td>0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9</td></tr>
<tr><td>based-literal</td><td>::=</td><td>base ! bdigit { bdigit }</td></tr>
<tr><td>base</td><td>::=</td><td>2 | 8 | 16</td></tr>
<tr><td>bdigit</td><td>::=</td><td>digit | A | B | C | D | E | F</td></tr>
</table>

Note no sign can be given to a based literal; an explicit two's complement representation is used. The constant is an signed integer value of the indicated type. If the value is too large for the indicated type, it is truncated on the left; excess bits are discarded. Sample signed integer constants are:

```
-1(SI32)
2!11111111(SI8)
8!177777(SI16)
16!FFFFFFFF(SI32)
```

An unsigned integer constant is similar, except there is no sign, and the type qualifiers are different.

<table>
<tr><td>unsigned-integer-constant</td><td>::=</td><td>(decimal-literal | based-literal) [‘(’ unsigned-integer-type ‘)’]</td></tr>
</table>

An unsigned integer can also be represented as a character constant. A character constant is a sequence of characters enclosed in quotes, followed by a type qualifer. We follow the C convention, and use `\` as an escape character, and allow `\nnn` to represent the the character whose three digit ASCII octal code is nnn.

<table>
<tr><td>character-constant</td><td>::=</td><td>‘ ‘ ‘ { cchar } ’ ’ ’ [‘(’ unsigned-integer-type ‘)’]</td></tr>
<tr><td>cchar</td><td>::=</td><td>any-char-except-slash-and-quote |
\’ ’ ’ |
\\ |
\ digit digit digit</td></tr>
</table>

A character represents the corresponding 8 bit ASCII code, and a sequence of characters is a left-to-right concatenation of the appropriate codes. If the resulting sequence is too long for the indicated type, it is truncated on the left. If it is too short, it is zero filled on the left. (Note: Is this what we want?)

The IL also supports a true character string literal, though it can be used only in initializations.

<table>
<tr><td>string-constant</td><td>::=</td><td>“ { schar } ”</td></tr>
<tr><td>schar</td><td>::=</td><td>any-char-except-slash-and-squote |
\’ ’ |
\\ |
\ digit digit digit</td></tr>
</table>

Strings are not null terminated, unless the last char in the string-constant is `\000`. Strings can be extended across multiple lines by ending a line with \\.

Pointer constants are similar to unsigned integers.

<table>
<tr><td>pointer-constant</td><td>::=</td><td>(decimal-literal | based-literal) [’(’ P32 ’)’]</td></tr>
</table>

(Note: The OS may need these. C supports them.)

A floating constant has the following form; the type qualifier can be omitted only if it is clear from context.

<table>
<tr><td>floating-constant</td><td>::=</td><td>[sign] digits . digits [E [sign] digits] [‘(’ floating-type ‘)’]
| [sign] digits E [sign] digits [‘(’ floating-type ‘)’]</td></tr>
<tr><td>digits</td><td>::=</td><td>digit { digit }</td></tr>
</table>

The constant is given the indicated type. Exponents too large or too small are an error. (Note: precision??)

1.6 Operands

Operands in an IL program are either variables, formal parameters, constants, temporaries, labels, or entry-point names.

- *Variables.* Variables must be declared before used; their declaration will state if memory is to be allocated for them. They can be scalars of any of the primitive types, or aggregates.
- *Temps.* Temps are named scalar quantities; their type is restricted to the primitive IL data types, with the addition of unsigned integers of any size between 1 and 32 bits (these are used to keep track of bit-fields when they are in temps). They are declared via TEMP-DECL. Temps come in two flavors: Temporaries created by the compiler to hold intermediate results, and user-specified variables which have been targeted to temporaries by the front-end. Temps are assumed to be able to live exclusively in registers. Their addresses can never be taken and their values are lost when the declaring procedure returns.
- *Parameters.* Parameters are passed by value and can be treated like local variables. Parameters must be declared.
- *Constants.* Constants operands are compile-time constants defined by oper-constant. They are either 32 or 64 bits long. They are not declared. Their literal representation is defined above. Load-time constants are not permitted as operands. They are only permitted in packet initializations, and are defined elsewhere. ⇐4.1
- *Labels.* Labels denote branch targets within a program. They are defined with the LABEL pseudo-op.
- *Entry-points.* Entry-points denote entries to routines. They are defined with the ENTRY pseudo-op.

The input operands of most IL operators are restricted to temps and oper-constants. As a convenience in describing input operands, we define an *rvalue* to be a temp or an oper-constant. ⇐4.1

rvalue	::=	temp | oper-constant

Certain high-level operations operate on complex values by using pairs of temps or rvalues. As a notational convenience, we define the following:

ctemp	::=	temp temp
crvalue	::=	rvalue rvalue

It should be clear from context when complex values are appropriate. C128 values consist of two F64 values. C64 values consist of two 32 bit values (including small integers) which are interpreted as F32 values. No coercion is performed; in the extremely unusual case where one of these values is not an F32 value, whatever bit pattern it contains will be used as if it were an F32 value.

1.7 Declarations

IL-1 programs declare referenced routines, temps, and variables. Labels are declared by defining them at the appropriate point in the program. ⇐4.1

There are two basic mechanisms for declaring variable names. One mechanism is used for declaring temps. The other mechanism is used for declaring all other variables, parameters, templates, etc. This latter mechanism requires the use of packets.

1.7.1 Routines

Every routine entry point which is referenced in an IL program (either in a direct call to the entry or by having its address taken with the RADDR operator) must be declared via a ROUTINE-DECL. This declaration causes an external request to be generated (if necessary), and also supplies the optimizer with useful information about the routine. ⇐4.2

routine-decl	::=	ROUTINE-DECL entry-name attribute-list
attribute-list	::=	{ attribute }
attribute	::=	EXTERN | DOES-NOT-MODIFY-ARG-BLOCK | DETERMINISTIC-RESULT | NO-SIDE-EFFECTS | ALWAYS-RETURNS | NEVER-RETURNS

⇐4.2

Entry-name is the name of the routine entry point being declared.

The various attributes supply useful information about the routine.

EXTERN indicates that an external request for the routine entry should be generated. However, there is no harm in specifying EXTERN for a routine entry which is later defined in the same module. ⇐4.2

DOES-NOT-MODIFY-ARG-BLOCK indicates that the argument block is unchanged by a call to this routine entry (this should always be true for FORTRAN routines). ⇐4.2

DETERMINISTIC-RESULT indicates that the return value is a function ⇐4.1 whose value is solely determined by the actual arguments given in the call. Exceptions are random number generators which keep their own seed, system status functions, etc. Note that this attribute cannot be used for almost all FORTRAN routines which take a by-reference argument, since the attribute applies to the actual argument passed. This is a constant pointer for by-reference arguments, so if DETERMINISTIC-RESULT were used here, it would imply that the result is independent of the value contained in the by-reference argument (i.e., the dereferenced value of the actual argument).

NO-SIDE-EFFECTS indicates that the routine has no side-effects, such as changing the value of a global, performing I/O, etc. (although it may change its actual arguments).

ALWAYS-RETURNS indicates that a call to the routine entry will always ⇐4.2 return (i.e., the routine won't suspend or terminate the process). Most library routines should have this attribute.

NEVER-RETURNS indicates that a call to the routine entry will never ⇐4.2 return, effectively terminating the process. This attribute may be used for the routine used to implement Fortran's STOP statement, or C's exit function.

Note that a recursive routine still needs a ROUTINE-DECL, either within the routine or at the module level.

1.7.2 Temps

Temps come in two flavors: Temporaries generated by the compiler to hold intermediate results, and variables which were specified by the user and which have been determined to be able to live in temporaries. The only reason the distinction is made is to preserve the fact that a temp was in fact introduced by the user. It should only make a difference for error messages, listings, debugging aids, etc.

Temps may have any primitive data type, but may also be unsigned integers of any size between 1 and 32 bits. Note, however, that all temps are regarded as being 32 bits long, except for F64 temps which are 64 bits long.

temp-data-type	::=	prim-data-type | general-ui-type
general-ui-type	::=	I1 | I2 | ... | I32

Note that unsigned-integer-type (any of I8, I16, or I32), which is part of prim-data-type, is a subset of general-ui-type.

Temps may never have their addresses taken. Their values are preserved over recursive procedure calls, with new incarnations being created upon each call. An incarnation is destroyed when a return is made from the procedure which created it. The front-ends should allocate variables to temps whenever possible, as an aid to the optimizations performed by Phase 2.

A temp declaration may only be supplied at the routine level. It includes the name of the temp, its primitive data type, and an indication of whether it was supplied by the user (temp-class REGISTER) or was generated by the front-end (temp-class TEMPORARY).

temp-decl	::=	TEMP-DCL name temp-data-type temp-class
temp-class	::=	(TEMPORARY | REGISTER)

Temps may be thought of as variables which have been allocated to registers, and must be representable in a machine register. The only unclear cases are small integers, i.e., I1, ..., I31, SI8, and SI16. The cases for I8, I16, SI8, and SI16 are described below; the cases for I1, ..., I31 are analogous to the cases for I8 and I16.

1. An I8 temporary is a 32-bit unsigned integer which contains an 8-bit unsigned quantity in its low-order 8 bits and 0's in its high-order 24 bits (256 legal values):

00000000	00000000	00000000	abcdefgh

2. An SI8 temporary is a 32-bit signed integer which contains an 8-bit signed quantity in its low-order 8 bits and whose high-order 24 bits are sign-extended (256 legal values):

aaaaaaaa	aaaaaaaa	aaaaaaaa	abcdefgh

3. An I16 temporary is a 32-bit unsigned integer which contains a 16-bit unsigned quantity in its low-order 16 bits and 0's in its high-order 16 bits (65536 legal values):

00000000	00000000	abcdefgh	ijklmnop

4. An SI16 temporary is a 32-bit signed integer which contains a 16-bit signed quantity in its low-order 16 bits and whose high-order 16 bits are sign-extended (65536 legal values):

aaaaaaaa	aaaaaaaa	abcdefgh	ijklmnop

1.7.3 Packets and Variables

Packets

All data objects (except temps) in an IL program are grouped into packets. All data objects which have a language-defined storage relationship to one another are in the same packet. Otherwise data objects are in unique packets. For example, all the members of a C structure or union must be in the same packet; similarly, all the variables in a FORTRAN common block, or in a FORTRAN equivalence must be in the same packet. Except for alignment restrictions, the compiler has complete freedom as to where to place packets in memory. However, the relative location of data objects within a packet is fixed by the packet definiton. The information associated with a single packet allows the compiler to determine the alias relationship (if any) between the various data objects in the packet. There is no alias relationship between data objects in different packets.

A packet declaration consists of a packet name, an alignment, a size in bytes, ⇐4.2
allocation information, loader information, a list of variables defined in the ⇐4.1
packet, and an optional initialization.

packet-decl	::=	DCL packet-name
		packet-alignment
		size
		allocation
		loader-info
		{ VAR var-decl }
		[initialization]

- *Packet-name.* A packet-name is provided for naming common blocks
  and other packets which require external names, and in any case is re- ⇐4.2
  quired to conveniently associate variables with their containing packets
  in output from the compiler.

- *Packet-alignment.* Packet-alignment indicates any restrictions on the
  alignment of the packet. It must be one of 0MOD1, 0MOD2, 0MOD4, ⇐4.2
  0MOD8, or 4MOD8. However, only TEMPLATE packets may have
  0MOD1 or 0MOD2 alignment; all other packets must have 0MOD4,
  0MOD8, or 4MOD8 alignment. Note that this implies that all non- ⇐4.1
  TEMPLATE packets are at least word-aligned. ⇐4.2

  The packet alignment may later be adjusted to a more restrictive (i.e., ⇐4.1
  larger) compatible alignment by Phase 2.

  Note that small integers with no required storage relationship to each other will be in separate packets and hence never in the same 32-bit word.

  There are additional alignment restrictions imposed by the allocation ⇐4.1
  and loader-info fields. In particular:

  - If allocation is F-COMMON then packet-alignment must be 0MOD8. ⇐4.2

  - If loader-info is ABLOCK then the preferred alignment of each
    top-level RECORD (i.e., each individual argument block) must
    be compatible with packet-alignment. In addition, the preferred
    alignment of each top-level RECORD must be 0MOD4 or 0MOD8,
    so packet-alignment must be 0MOD4 or 0MOD8. ⇐4.3

    The offset of each top-level RECORD within the ABLOCK packet
    must be 0. ⇐4.3

    If a given top-level RECORD contains a variable whose preferred alignment is 0MOD8, then the preferred alignment of that top-level RECORD must be 0MOD8. Otherwise it must be 0MOD4. ⇐4.3

Within each top-level RECORD, each top-level component of that RECORD corresponds to an actual argument. The top-level components of the RECORD are allocated sequentially, in the order of the corresponding actual arguments, as follows: ⇐4.3

- ∗ If the argument variable is a SCALAR whose primitive data type is I8 or SI8, first skip ahead to the next 0MOD4 offset within the RECORD (if not already at such an offset). Then skip ahead to the next 3MOD4 offset within the RECORD and allocate the argument at that offset. Note that this case doesn't arise in C, since small integer valued expressions are coerced to normal integer valued expressions when used as actual arguments. ⇐4.3
- ∗ If the argument variable is a SCALAR whose primitive data type is I16 or SI16, first skip ahead to the next 0MOD4 offset within the RECORD (if not already at such an offset). Then skip ahead to the next 2MOD4 offset within the RECORD and allocate the argument at that offset. Note that this case doesn't arise in C, since small integer valued expressions are coerced to normal integer valued expressions when used as actual arguments. ⇐4.3
- ∗ If the preferred alignment of the argument variable is 0MOD8, first skip ahead to the next 0MOD8 offset within the RECORD (if not already at such an offset). Then allocate the argument at that offset. ⇐4.3
- ∗ If none of the above cases apply, first skip ahead to the next 0MOD4 offset within the RECORD (if not already at such an offset). Then allocate the argument at that offset. ⇐4.3

– If allocation is PARAM then packet-alignment must be the preferred alignment of the sole top-level component of the packet (i.e., the top-level RECORD). In addition, the preferred alignment of the top-level record must be 0MOD4 or 0MOD8, so packet-alignment must be 0MOD4 or 0MOD8. ⇐4.3

  The offset of the sole top-level RECORD within the PARAM packet must be 0. ⇐4.3

  If the top-level RECORD contains a variable whose preferred alignment is 0MOD8, then the preferred alignment of the top-level RECORD must be 0MOD8. Otherwise it must be 0MOD4. ⇐4.3

Within the top-level RECORD, each top-level component of the RECORD corresponds to a formal argument. The top-level components of the RECORD are allocated sequentially, in the order of the corresponding formal arguments, as follows: ⇐4.3

  * If the argument variable is a SCALAR whose primitive data type is I8 or SI8, first skip ahead to the next 0MOD4 offset within the RECORD (if not already at such an offset). Then skip ahead to the next 3MOD4 offset within the RECORD and allocate the argument at that offset. ⇐4.3
  * If the argument variable is a SCALAR whose primitive data type is I16 or SI16, first skip ahead to the next 0MOD4 offset within the RECORD (if not already at such an offset). Then skip ahead to the next 2MOD4 offset within the RECORD and allocate the argument at that offset. ⇐4.3
  * If the preferred alignment of the argument variable is 0MOD8, first skip ahead to the next 0MOD8 offset within the RECORD (if not already at such an offset). Then allocate the argument at that offset. ⇐4.3
  * If none of the above cases apply, first skip ahead to the next 0MOD4 offset within the RECORD (if not already at such an offset). Then allocate the argument at that offset. ⇐4.3

– If allocation is TEMPLATE, let the preferred alignment of the sole top-level component of the packet be 0MODn. Then packet-alignment must be 0MODm, where m is the largest power of two such that: ⇐4.3

  1. $m \leq n$ ⇐4.3
  2. All VOTGEN and scalar memory reference operations which use the TEMPLATE are guaranteed to be through base pointers whose values are 0 (mod m). ⇐4.3

  The offset of the sole top-level component of the TEMPLATE packet must be 0. ⇐4.3

– If none of the above special cases apply, the front-end should specify 0MOD4 alignment for packets which don't contain any F64 variables. For packets which contain F64 variables, an attempt should be made to specify an alignment which minimizes the number of unaligned F64 variables. However, note that for global variables a consistent alignment must be used in all declarations of the same packet. ⇐4.2

- *Size.* Size is the size of the packet in bytes.

size	::=	I32-constant
		| *

⇐4.1

More precisely, the size of the packet is:

```
(addr.lastb - addr.firstb) + 1
```

where addr.lastb is the address of the last byte in the packet, and addr.firstb is the address of the first byte in the packet. Because of alignment considerations there may be "gaps" in a packet containing mulitiple variables. Also note that the size and alignment of a packet ⇐4.2 are independent, and in general the size of a packet need not be a multiple of the base of the packet alignment. Phase 2 will allocate any padding needed between packets.

A size is an unsigned integer constant, or a * if unknown. The size of ⇐4.2 a packet should be specified as an unsigned integer constant whenever possible. An unknown packet size may only be used if the allocation of the packet is TEMPLATE or if the loader-info of the packet is EXTERN.

An unknown size should only be needed for packets used to reference arrays of unknown size. For example, this can happen in TEMPLATE packets used to reference FORTRAN assumed-size or adjustably dimensioned arrays. It can also happen in EXTERN packets in C when an external request is made for an array without specifying the size of the array. ⇐4.2

However, an unknown size should not be used for FORTRAN common blocks. Instead, each routine should specify the size of the common block as declared by that routine. In the case of blank common, it is up to the linker to find the largest size specified and to allocate that much storage for the common block. ⇐4.2

Note that size, when known, is a redundant piece of information, which ⇐4.2 may be computed as follows:

```
MAX ( {var.offset + var.size : var is a top-level
                                member of the packet} )
```

- *Allocation.* Allocation is one of the following possibilities.

allocation	::=	F-COMMON
		| C-COMMON
		| STATIC
		| ESTATIC
		| KSTATIC
		| AUTO
		| PARAM
		| TEMPLATE

⇐4.2

F-COMMON means the packet is a FORTRAN common block. If the block is uninitialized, loader-info must be EXTERN, and the linker must resolve all references to the packet, allocating storage if necessary. If the block is initialized, loader-info must be DEFINED, and Phase 2 will explicitly allocate and initialize the block. A common block may be initialized in at most one place, in which case all other references to the common block will be resolved to the address of the initialized common block by the linker. In all cases, static storage will be used to allocate the common block. Note that Phase 2 may compatibly adjust the packet alignment supplied by the front end. ⇐4.2

C-COMMON is basically the same as F-COMMON. However, Phase 2 may adjust the alignment differently than in the F-COMMON case. ⇐4.3 This allocation is provided to permit more relaxed rules for declaring and defining external C variables. ⇐4.2

STATIC means storage for the packet is fixed for the lifetime of the program, and that values are preserved between procedure calls.

ESTATIC is valid only at the routine level. It is the default FORTRAN local variable allocation. Basically, it is the same as STATIC except that an ESTATIC variable is undefined upon entry to a routine unless it is initialized (i.e., in a DATA statement). In this case, the only defined value it can have upon entry to the routine is the value given in the DATA statement.

KSTATIC is used for read-only constant data which must reside in main memory. Unless it is EXTERN, it must be explicitly initialized (in the EXTERN case, it is assumed to be statically initialized elsewhere). It is used for compiler-generated constants such as numbers or character strings, and for constant argument blocks in FORTRAN. We might eventually want to provide a source-level "readonly" qualifier for use in our own C code (as well as the operating system's C code). This allows better optimization, trace scheduling, and allocation. Note that some C compilers provide this with the "const" specifier.

AUTO means storage for the packet is allocated automatically on entry to a routine and freed on exit.

PARAM means the storage contains the parameters for a routine; this storage may be automatic or static, depending on the requirements of the source language and the actual parameter passing mechanism. A ⇐4.1
PARAM packet must contain exactly one top-level component, which must be a RECORD.

TEMPLATE means the packet has no storage allocated for it; rather the packet serves as a template for a block of storage referenced by a pointer.

- *Loader information.* Loader-info is one of the following possibilities.

loader-info	::=	EXTERN
		| DEFINED
		| INTERN
		| ABLOCK

⇐4.2

EXTERN means storage for the packet is not explicitly allocated by this declaration. If the allocation is F-COMMON or C-COMMON, the linker will allocate the storage itself, if necessary. Otherwise, the program must be linked with some other declaration of the packet which either explicitly or implicitly (i.e., through F-COMMON or C-COMMON) causes storage to be allocated for the packet. ⇐4.2

DEFINED means the storage for the packet is explicitly allocated here, and the address of the storage must be supplied to the linker to satisfy possible external requests for the packet. ⇐4.2

INTERN means the packet is accessible only to the current module.

ABLOCK is basically the same as INTERN. It is used to declare argument blocks for calling other routines. The address of this RECORD ⇐4.3

is given to the routine being called.

The following table describes the valid allocation / loader-info combinations. Note that many accepted combinations will not be generated by the front-ends. Such combinations are listed here to support future extensions and to promote orthogonality of design in the back-end. It is up to the individual front-ends to check for illegal storage class specifications, even if they are allowed in this table. The back-end must assume that all combinations indicated as legal in the table can occur.

Each entry is of the form "| X | Y |", where X indicates what is legal at the module level, and Y indicates what is legal at the routine level. The individual values are:

-	illegal combination
X	legal, INIT illegal
R	legal, INIT required
O	legal, INIT optional

	loader-info							
allocation	EXTERN		DEFINED		INTERN		ABLOCK	
F-COMMON	X	X	R	R	-	-	-	-
C-COMMON	X	X	R	R	-	-	-	-
STATIC	X	X	O	O	O	O	-	-
ESTATIC	-	-	-	-	-	O	-	O
KSTATIC	X	X	R	R	R	R	-	R
AUTO	-	-	-	-	-	X	-	X
PARAM	-	-	-	-	-	X	-	-
TEMPLATE	-	-	-	-	X	X	-	-

⇐4.2

- *Var-decl.* Var-decl is defined in the next section. However, there is a ⇐4.1
restriction. If allocation is PARAM or TEMPLATE, then there must ⇐4.3
be exactly one top-level variable in the packet. The intent is that all aliasing will be done at the RECORD level, except for COMMON and EQUIVALENCE aliasing, since this type of aliasing can violate the alignment rules of RECORDs. This restriction is therefore more permissive than it might be.

  Also, if allocation is PARAM or loader-info is ABLOCK, then each top-level variable in the packet must be a RECORD. As mentioned above, in the case of PARAM packets there will be only one top-level variable. ⇐4.3

- *Initialization.* Initialization is a sequence of offsets paired with initialization items.

intialization	::=	INIT { offset repeat-count data }
offset	::=	I32-constant
repeat-count	::=	I32-constant
data	::=	init-constant | load-time-constant
load-time-constant	::=	variable-address | routine-address
variable-address routine-address	::=	VADDR variable-name variable-offset | RADDR routine-name
variable-offset	::=	SI32-constant

⇐4.1

Offset is the offset of the beginning of the data from the start of the packet, in bytes. The data is stored in memory starting at the indicated offset. ⇐4.1

Repeat-count is the number of times the data should be duplicated. Repeated items are packed in memory. Note that init-constants each have a well-defined size, and load-time-constants are all 32 bits. ⇐4.1

The list of offset/repeat-count/data tuples must be sorted by offset in increasing order. It is illegal for any one byte to be initialized by more than one tuple. Note, however, that different bytes in a given word may be initialized by different tuples. It is illegal for any byte outside of a packet to be initialized by that packet. ⇐4.1

If a routine-address is used in an initialization, then routine-name must be the name of a routine entry point declared in a ROUTINE-DECL. The load-time-constant used in the initialization is the address of that ⇐4.2
routine entry point. ⇐4.1

If a variable-address is used in an initialization, then variable-name must be the name of a variable in some static packet (one of F-COMMON, C-COMMON, STATIC, ESTATIC, or KSTATIC). The ⇐4.2
load-time-constant used in the initialization is the sum of the address of variable-name and variable-offset. It should be noted that regardless of the value of variable-offset, the load-time-constant is still considered to point at variable-name. For example, if we have a packet containing two consecutive integers A and B, and initialize a pointer to be

"VADDR B -4", then even though it will in reality be equal to the address of A, it cannot be legally used to reference A. In other words, it may not be legally dereferenced as it stands, but may be validly used to reference B if 4 is added to it. ⇐4.1

The types of the constants used for initialization can be significant (other than indicating the size). Specifically, in Fortran an SI32 variable is assumed to be initially undefined if initialized with an F32-constant. This could happen if the SI32 variable were aliased with an F32 variable. These aliasing rules are, unfortunately, language-specific. The front-ends should be aware of the correct rules for their language. Note that load-time-constants all have type P32. ⇐4.1

A packed collection may be initialized by initializing the entire packed collection with an I32-constant. ⇐4.1

An initialization may only be given for packets whose allocation is F-COMMON, C-COMMON, STATIC, ESTATIC, or KSTATIC, and ⇐4.2
whose loader-info is not EXTERN.

(Note: If the storage required proves to be too much for large Fortran arrays initialized with implied DO-loops, then we may wish to add a more concise way of specifying such initializations. However, the current scheme should be adequate for the time being.) ⇐4.1

Variables

Variables in IL are named data objects. A variable is defined within one packet. If the variable has no language-defined relationship with any other variables, it will be the only variable defined in the packet. Otherwise there will be more than one variable in a packet. Note that temps are not regarded as variables here.

A variable declaration consists of a name, a size, an offset, an esize, optional dimension information, optional argument information, and type informa- ⇐4.2
tion. The type of a variable is either scalar, record, or a packed collection of fields. By including dimension information, the variable is declared to be an array of objects of the given type. Argument information is present only for certain variables in an ABLOCK packet (see the description of arg-info below). ⇐4.2

```
var-decl  ::=  name
               size
               offset
               esize
               ( dimension-info | NIL )
               ( arg-info | NIL )
               type-info
```

⇐4.2

- *Name.* Name is the name of the variable. It is a sequence of characters, following the rules of IL names.

- *Size.* Size is syntactically as defined for packets, and gives the size of the variable in bytes. This size does not take into account any possible inherited dimension information of any parent records, but it does take into account any dimension information directly associated with the variable. ⇐4.2

  For user variables in C, size should be exactly what sizeof would give (when known). An unknown size should only be used for arrays of unknown size. An unknown size may only be used for variables whose packet has an unknown size (see the restrictions for unknown packet size in the previous section). ⇐4.2

- *Offset.* Offset is the offset in bytes of the first byte of the variable from the first byte of the packet. For variables in (possibly nested) arrays, we calculate the offset as if we were only considering the first element of each array.

  ```
  offset  ::=  I32-constant
  ```

  ⇐4.1

- *Esize.* Esize is an I32-constant, and gives the size of the variable in bytes. This size does not take into account any possible inherited dimension information of any parent records, nor does it take into account any dimension information directly associated with the variable. Since it must be constant, it is implicit that no record may contain a variable of unknown size. For a variable which is not dimensioned directly, esize and size must be the same. Otherwise, size should be the product of esize and the number of elements in the array (when the latter is known). ⇐4.2

esize	::=	I32-constant

⇐4.2

For user variables in C, esize should be exactly what sizeof would give if the directly associated dimension information were absent. ⇐4.2

In addition, if the variable is a directly dimensioned record then the size of the record (i.e., esize) must be a multiple of the base of the ⇐4.2 preferred alignment of the record. ⇐4.3

- *Dimension information.* Dimension-info is optional, and if included declares the variable to be an array of objects of the indicated type. If dimension-info is omitted, the variable is a single object of the indicated type.

dimension-info	::=	ARRAY dim-entry-list
dim-entry-list	::=	dim-entry { , dim-entry }
dim-entry	::=	dim-bound dim-bound
dim-bound	::=	rvalue | *

⇐4.1

There is a dim-entry for each dimension of the array. It gives the lower and upper bounds for the dimension. This information is provided for ⇐4 use by the assertion facility. See discussion of array loads and stores.

Each dim-bound which is not * must have an SI32 type, i.e., it must be either an SI32-constant or a temp of type SI32. The bounds are signed integer constants for all variables not in TEMPLATE packets ⇐4.1 (with the exception of the upper bound of the first dimension of an EXTERN packet, which may be unknown, i.e., *). If the variable is ⇐4.2 in a TEMPLATE packet, the dimension bounds may not be known at compile time. If this is the case, integer valued temps can be used. If temps are used, they are evaluated on entry to the routine declaring the template, and they cannot be changed in the body of the routine. We count on dead code removal to remove these operations if the temps are not used in the body of the routine. The upper bound in the first dimension of a TEMPLATE variable may be unspecified, in which case a * is used. See the discussion of F77 array parameters in the Fortran translation guide.

Arrays are stored in row major order. Arrays must always be packed.

- *Argument information.* Arg-info, if present, supplies information about the variable when it is used as part of an argument block. This is a very special case. Each argument block must be declared as a top-level RECORD in an ABLOCK packet. The top-level members of the ⇐4.3
RECORD are the arguments to the routine. These top-level members are the only variables which can have arg-info.

arg-info	::=	arg-info-item { , arg-info-item }
arg-info-item	::=	PTR-TARGET-NOT-MODIFIED | PTR-NOT-REMEMBERED

  PTR-TARGET-NOT-MODIFIED applies to arguments which are pointers, and indicates that the routine will not modify the target of the pointer. ⇐4.2

  PTR-NOT-REMEMBERED applies to arguments which are pointers, and indicates that the routine will not save the value of the pointer after it returns. ⇐4.2

- *Type information.* Type-info defines the type of the variable. The type is either scalar, record, or packed collection. In addition, each variable has a preferred alignment, as described below in the descriptions of ⇐4.1
the individual variable types. ⇐4.3

type-info	::=	scalar-info | record-info | packed-info

  - Scalar-info declares the type of the variable to be a scalar of the indicated primitive data type.

scalar-info	::=	SCALAR prim-data-type

    Prim-data-type is one of the IL primitive data types. ⇐4.1
    The preferred alignment is implicit from the prim-data-type. ⇐4.3

  - Record-info declares the type of the variable to be a record. It includes a list of var-decl's describing the components of the record.

record-info	::=	RECORD preferred-alignment { COMPONENT var-decl } END-RECORD
preferred-alignment	::=	0MOD1 | 0MOD2 | 0MOD4 | 0MOD8

⇐4

The components of a record are considered variables in the IL, though the IL does know they are components of a record. The offsets that define them are from the start of the packet, not from the start of the record. The unique name rule also applies to components, and they can be referred to by name directly, without being qualifed by the name of the enclosing record. ⇐4.1

Each component of the record will have its offset defined in terms of the containing packet. Assume the preferred alignment of the component variable is 0MODn. If we compute its offset with respect to the beginning of the record (i.e., by subtracting the record offset from the component variable's offset), the result must be 0 (mod n). ⇐4.1

Preferred-alignment is determined by Phase 1 and is supplied to Phase 2 in the record-info. The preferred alignment of each component variable must be compatible with the preferred alignment of the record. However, in itself this restriction is insufficient for Phase 2 to determine the preferred alignment of the record. This is because Phase 1 may have additional information about the record of which Phase 2 is unaware. Indeed, there may even be some private Phase 1 options which permit the user to control the layout and alignment of certain user structures. ⇐4.3

- Packed-info declares the type of the variable to be a packed collection of fields.

packed-info	::=	PACKED { FIELD field-decl } END-PACKED
field-decl	::=	name first-bit flength
first-bit	::=	I32-constant
flength	::=	I32-constant

⇐4

Name is the name of the field. First-bit is the offset in bits of the ⇐4.1
field from the start of the packed data object; bit numbering is zero based, and from left to right. Flength is the length of the field in bits, and must be in the range 1..32. ⇐4.2

The individual field elements are intended to be interpreted as unsigned integers.

Packed collections are constrained to be 32 bits in length. Their preferred and acceptable alignments are both 0MOD4. ⇐4.1

1.7.4 Some Sample Translations.

We include some sample translations of source language declarations to IL declarations.

1. Scalars, arrays, common blocks, equivalences. Consider the following FORTRAN declarations.

   ```
   INTEGER A,B(8)
   DOUBLE PRECISION C,D
   EQUIVALENCE (B(5),D)
   COMMON /TEST/A,B
   DATA C/5.0/
   ```

   A,B,D are related through the COMMON and EQUIVALENCE statements, and therefore the corresponding IL variables must be in the same packet. C is initialized, and therefore must be in memory. The IL declarations are:

   ```
                           ; packet for C
   DCL PACK1 0MOD8 8 ESTATIC INTERN    -
       VAR C 8 0 8 NIL NIL SCALAR F64  -
       INIT OFFSET 0 5.0(F64)

                           ; packet for common block TEST
   DCL TEST 0MOD8 36 F-COMMON EXTERN   -
       VAR A 4 0 4 NIL NIL SCALAR SI32 -
       VAR B 32 4 4                    -
           ARRAY 1 8                   -
           NIL SCALAR SI32             -
       VAR D 8 20 8 NIL NIL SCALAR F64
   ```

   The alignment of TEST forces D to be aligned 4MOD8, so D is misaligned for an F64 value. ⇐4.2

2. Reference parameters. Consider the declaration of the following FORTRAN parameter.

```
SUBROUTINE FOO(N,M)
INTEGER N,M
```

IL uses call by value; FORTRAN call-by-reference parameters are represented in the IL as pointers passed by value. However, the FORTRAN declarations of the parameters allow us to create IL TEMPLATES which describe the values referenced by the pointers.

All the parameters for a routine are declared in one packet (see section on routines). ⇐4.1

```
                        ; packet for parameters
DCL PACK1 OMOD4 8 PARAM INTERN          -
    VAR N 4 0 4 NIL NIL SCALAR P32      -
    VAR M 4 4 4 NIL NIL SCALAR P32

                        ; template for actual referenced by N
DCL PACK2 OMOD4 4 TEMPLATE INTERN       -
    VAR N' 4 0 4 NIL NIL SCALAR SI32

                        ; template for actual referenced by M
DCL PACK3 OMOD4 4 TEMPLATE INTERN       -
    VAR M' 4 0 4 NIL NIL SCALAR SI32
```

3. Records. Consider the following C declarations, which are local to some procedure.

```
struct apartment {
    int     floor;
    char    letter;
};

struct apartment    b, ba[1000], *aptr, *bptr;
```

In the IL, there will be several distinct declarations derived from the APARTMENT record type. The declarations of B and BA each require local packets which have components corresponding to the fields of APARTMENT. The declaration of APTR and BPTR as pointers requires an IL TEMPLATE corresponding to the APARTMENT. FLOOR and LETTER are related through the type declaration of APARTMENT, and each pair of IL variables corresponding to them must always appear in the same packet.

To support aggregate assignment and parameter passing, each packet corresponding to APARTMENT must have identical relative offsets for floor and letter. In addition, the acceptable alignment for the apartment record must be compatible with the alignment which is actually used.

```
                        ; packet for APARTMENT template.
DCL PACK1 OMOD4 8 TEMPLATE INTERN                           -
    VAR apartment 8 0 8 NIL NIL                             -
        RECORD OMOD4                                        -
            COMPONENT floor1  4 0 4 NIL NIL SCALAR SI32     -
            COMPONENT letter1 1 4 1 NIL NIL SCALAR SI8      -
            END-RECORD

                        ; packet for APTR
DCL PACK2 OMOD4 4 AUTO INTERN                               -
    VAR aptr 4 0 4 NIL NIL SCALAR P32

                        ; packet for BPTR
DCL PACK3 OMOD4 4 AUTO INTERN                               -
    VAR bptr 4 0 4 NIL NIL SCALAR P32

                        ; packet for B
DCL PACK4 OMOD4 8 AUTO INTERN                               -
    VAR b 8 0 8 NIL NIL                                     -
        RECORD OMOD4                                        -
            COMPONENT floor2  4 0 4 NIL NIL SCALAR SI32     -
            COMPONENT letter2 1 4 1 NIL NIL SCALAR SI8      -
            END-RECORD

                        ; packet for BA
DCL PACK5 OMOD4 8000 AUTO INTERN                            -
    VAR ba 8000 0 8                                         -
```

```
            ARRAY 0 999                                  -
            NIL                                          -
            RECORD 0MOD4                                 -
                COMPONENT floor3  4 0 4 NIL NIL SCALAR SI32 -
                COMPONENT letter3 1 4 1 NIL NIL SCALAR SI8  -
                END-RECORD
```

Notice we can refer to FLOOR2 unambiguously without giving context information.

4. Fields. Consider the C declaration:

```
static struct {
    unsigned is-keyword : 1;
    unsigned is-extern  : 1;
    unsigned is-static  : 1;
} flags;
```

It is translated to the following IL declaration. The C manual states that C fields are packed into a machine integer. Therefore, flags is aligned 0MOD4 and has size 4, rather than byte alignment and size 1.

```
                        ; packet for flags
DCL PACK6 0MOD4 4 STATIC INTERN             -
    VAR flags 4 0 4 NIL NIL                 -
        RECORD 0MOD4                        -
            COMPONENT xflags 4 0 4 NIL NIL  -
                PACKED                      -
                    FIELD is-keyword 0 1    -
                    FIELD is-extern  1 1    -
                    FIELD is-static  2 1    -
                    END-PACKED              -
            END-RECORD
```

1.8 Referencing objects in memory.

IL-1 provides several high-level addressing mechanisms for referencing memory. A scalar can be referenced by its name; an array element can be referenced by the name of the array and a list of indices; and a field can be referenced by its name.

All variables are at some offset within a packet. To understand how to reference variables, we must first have a model of how packets are addressed. If the packet is static, its address is a constant supplied by the loader. If the packet is AUTO, its address is an offset from the stack pointer *sp*. If the packet is PARAM, its address is is given by the argument pointer *ap*. If the packet is TEMPLATE, its address is given by a pointer used in the reference.

There is a separate set of operators for performing memory references on variables in TEMPLATE packets. A base pointer to a packet is a required ⇐4
operand in references to TEMPLATE packets; the pointer must address an incarnation of the packet. A base pointer is never an operand in other memory references. Phase 2 knows about the stack pointer and argument pointer, and can supply them as necessary when expanding memory references into IL-2 for the code generator.

1.8.1 Loads and Stores

The only operators which can access memory are LOADs and STOREs. There are a variety of load and store operators: operators to load and store scalars by name (SLOAD, STLOAD, SSTORE, STSTORE), array elements by name (ALOAD, ATLOAD, ASTORE, ATSTORE), and bit fields by name (FLOAD, FTLOAD, FSTORE, FTSTORE, FALOAD, FATLOAD, FASTORE, FATSTORE).

All of the generic load and store operators are qualified as much as necessary to indicate the effects of the operation. For straight 32 or 64 bit operations, values are transferred directly between temps and memory without any conversion necessary. When storing a small integer of some temp-data-type to memory, the 32 bit value is truncated. If the data type is signed, an overflow ⇐4
can be detected. When loading a small integer of some prim-data-type into a temp, sign extension is performed on signed values, and unsigned values are zero-filled. Therefore, these loads and stores of small integers are qualified by prim-data-type. The exceptions are bit field operators, which are qualified by the number of bits being loaded or stored.

load-qualifier	::=	I8 | SI8 | I16 | SI16 | 32 | 64
store-qualifier	::=	I8 | SI8 | I16 | SI16 | 32 | 64

⇐4

Scalars

A scalar variable is referenced by name with SLOAD, STLOAD, SSTORE, or STSTORE.

scalar-load	::=	SLOAD.load-qualifier temp name
		| STLOAD.load-qualifier temp name bp

⇐4

Name is the name of an IL scalar variable. It must have a prim-data-type, and must be of the size indicated by load-qualifier. Temp is loaded with the ⇐4.1 value of the variable; sign-extension is performed for the SI8 and SI16 cases. The I8 and I16 cases are zero-padded.

STLOAD is used if the variable is in a TEMPLATE packet; SLOAD is used otherwise. Bp is a temp whose value is a pointer; it must point at an incarnation of the template.

scalar-store	::=	SSTORE.store-qualifier name rvalue
		| STSTORE.store-qualifier name rvalue bp

⇐4

In a store, the rvalue is stored in the named variable. STSTORE is used if the variable is in a TEMPLATE, SSTORE otherwise. Name must have a prim-data-type, and must be of the appropriate size.

For example, consider an integer variable I in an AUTO packet.

```
DCL 26 NIL OMOD4 4 AUTO INTERN          -
    VAR I 4 0 NIL NIL SCALAR SI32
```

To load I into t0 we write.

```
SLOAD.32 t0 I
```

The implicit address calculation is

```
sp + offset(packet(I)) + offset(I)
```

⇐4

where `sp` is the stack pointer, `offset(packet(I))` is the offset of I's packet (i.e., packet 26) from the start of the activation record, and `offset(I)` is the offset of I from the start of packet 26.

Recall components of records can be loaded and stored as scalar variables.

Arrays

All operators which perform array index calculations have an additional qualifier which indicates what the index expression should be multiplied (scaled) by to obtain a byte index. This allows us to take fuller advantage of the Saturn's address calculation instructions.

sval	::=	1 | 2 | 4 | 8

In order to directly reference an element of an array, a "virtual-origin" must first be computed. This is done with the VOGEN and VOTGEN operators.

vo-oper	::=	vogen-oper | votgen-oper
vogen-oper	::=	VOGEN.sval temp name index-list
votgen-oper	::=	VOTGEN.sval temp name bp index-list
index-list	::=	{ stride index }
stride	::=	rvalue
index	::=	rvalue

⇐4

Stride and index must be 32 bit quantities. They are interpreted as signed integers. Temp must be a 32 bit quantity. It is interpreted as a pointer. Name is the name of an array variable (or a component of a record in an array of records). VOTGEN is used when name is in a TEMPLATE packet; bp is required to point at an incarnation of the array.

The result of a VOGEN or VOTGEN operation is that temp is assigned the value ⇐4.1

$$\{0, sp, ap, bp\} + offset(packet(name)) + offset(name) - sval \times \sum_{i=1}^{n} stride_i \times index_i$$

⇐4
⇐4.1

Only one of $\{0, sp, ap, bp\}$ is added, depending on the allocation of the variable. The intent is that $stride_i$ will indicate the distance (in the appropriate units) between elements of the dimension whose lower bound is given by $index_i$. See examples below. ⇐4.1

VOTGEN permits us to fold the base address passed for FORTRAN by-reference arrays into the virtual origin of the array.

The strides and lower bounds of an array are always routine invariant in C and F77, as are the ap and sp. Therefore, for static, automatic, and by-reference arrays, the virtual origin is either a constant or routine invariant, and it can be calculated once upon entry to a routine. (For the less common case of a pointer moving between arrays, we do less well, calculating the virtual origin for each value of the pointer. We can add a third virtual origin operator if this is a problem.)

Array type	Virtual origin
static	load-time constant
automatic	compile-time constant + *sp*
by-reference	
w/ constant bounds	compile-time constant + arg-value
w/ adjustable bounds	routine-invariant + arg-value

⇐4

Referencing array elements. An array element is referenced with ALOAD, ATLOAD, ASTORE, or ATSTORE.

array-load	::=	array-load-op.sval.load-qualifier temp name vo index-list
array-load-op	::=	ALOAD | ATLOAD
vo	::=	rvalue

⇐4

Name is the name of an array variable (or a component of a record in an array of records – see examples below). Vo should be a virtual origin generated by VOGEN or VOTGEN. No base pointer is present for either ALOAD or ATLOAD. The stride-index pairs in the index list must be in one-to-one ⇐4 correspondence with the the dimensions declared for the array. This will permit the dimension information to be used in assertions.

This restriction means the front end cannot be clever in cases such as `A(I+1)`, where TJK has suggested the front end could create two stride/index pairs for this dimension when referencing this element, i.e., "stride I stride 1". Rather, all optimizations of this sort must be internal to Phase 2.

The address computation performed is

$$vo + sval \times \sum_{i=1}^{n} stride_i \times index_i$$

⇐4

Note that $stride_i$ and $index_i$ are the elements of index-list. The intent is that $stride_i$ will indicate the distance (in the appropriate units) between elements of the dimension whose index is given by $index_i$.

Temp is loaded with the referenced array element as in the scalar case. ATLOAD is used if the variable is in a TEMPLATE packet; ALOAD is used otherwise.

Note that name isn't needed to perform this operation. However, it is still provided to supply information to the optimizer.

An example:

```
DCL 7 NIL OMOD8 96 STATIC INTERN    -
    VAR A 96 0                      -
        ARRAY 4     1 2, 1 4, 1 3   -
        NIL SCALAR F32
```

To obtain a virtual origin for A, we could use the following (we would do this upon entry to the routine):

```
VOGEN.4 vo A NIL 12 1 3 1 1 1
```

Then to load A(t1,t2,t3) into t0 we could write:

```
ALOAD.4.32 t0 A vo 12 t1 3 t2 1 t3
```

An array store is similar to an array load and a scalar store. The rvalue is stored in the indexed element of the named variable. ATSTORE is used if the variable is in a TEMPLATE packet; ASTORE otherwise.

array-store	::=	array-store-op.sval.store-qualifier name rvalue vo index-list
array-store-op	::=	ASTORE | ATSTORE

⇐4

Fields

Fields can be referenced by name like scalars. Recall fields are bit fields in a packed collection; components of records are referenced with the scalar loads and stores.

field-load	::=	FLOAD.field-qualifier temp name | FTLOAD.field-qualifier temp name bp
field-qualifier	::=	1 | 2 | ... | 32

⇐4

Temp must be a 32 bit quantity (recall that small integer temps are regarded as special 32 bit quantities). Name is the name of an IL field whose size is field-qualifier bits. The field is extracted and the result stored (right-justified) in temp. The remainder of temp is zero-filled. FTLOAD is used if the variable is in a TEMPLATE packet; FLOAD is used otherwise. If the variable is in a template, then a base pointer bp must be given, and it must point at an incarnation of the template.

field-store	::=	FSTORE.field-qualifier name rvalue
		| FTSTORE.field-qualifier name rvalue bp

⇐4

Name is the name of an IL field whose size is field-qualifier bits. Rvalue must be a 32 bit quantity. In a field store, rvalue is truncated on the left as appropriate and stored in the named field. FTSTORE is used if the variable is in a TEMPLATE packet; FSTORE otherwise.

Fields in an array of packed collections can be referenced similarly to an array element.

field-array-load	::=	field-array-load-op.sval.field-qualifier temp name vo index-list
field-array-load-op	::=	FALOAD | FATLOAD

⇐4

Temp must be a 32 bit quantity. Name is the name of an IL field whose size is field-qualifier bits and which is a member of an array of packed collections. Address calculation is performed as in the standard array case. The bit offset of the field is taken from the name. The field is extracted and the result stored (right-justified) in temp. The remainder of temp is zero-filled. FATLOAD is used if the variable is in a TEMPLATE packet; FALOAD is used otherwise. Note that the name of the packed collection is used to ⇐4.1
generate the virtual origin in the VOGEN or VOTGEN operation.

A field array store is similar to a field array load and a field store. Rvalue is truncated on the left if necessary, and stored in the selected field. FATSTORE is used if the field is a TEMPLATE; FASTORE otherwise.

field-array-store	::=	field-array-store-op.sval.field-qualifier name rvalue vo index-list
field-array-store-op	::=	FASTORE | FATSTORE

⇐4

For example, suppose flag1 is a field in a three dimensional array of packed collections.

```
DCL 7 NIL OMOD4 96 STATIC INTERN          -
    VAR flags 96 0                        -
        ARRAY 4     1 2, 1 4, 1 3         -
        NIL                               -
        RECORD OMOD4                      -
            COMPONENT xflags 4 0 NIL NIL  -
                PACKED                    -
                    FIELD flag1 0 1       -
                    FIELD flag2 1 1       -
                    FIELD flag3 2 1       -
                    END-PACKED            -
            END-RECORD
```

⇐4.1

Then to reference `flags(t1,t2,t3).flag1` we could write ⇐4.1

```
FALOAD.4.1 t0 flag1 vo 12 t1 3 t2 1 t3
```

where vo has been previously generated with a VOGEN operation (applied to xflags).

1.8.2 Calculating addresses

The IL also provides operations which evaluate the address of a named variable, array element, routine, or program label into a temp.

Addresses of scalars.

scalar-addr	::=	SADDR temp name
		| STADDR temp name bp

⇐4

Temp must be a 32 bit quantity. Name is the name of an IL variable. Note that name may be a record, array, or packed collection as well as a scalar of some prim-data-type. Temp is assigned the address of the variable. If the variable is dimensioned, the address of the first element of that array is used. STADDR is used when the variable is in a TEMPLATE packet; SADDR is used otherwise. If the variable is in a template, then the base pointer bp must be given, and it must point at an incarnation of the template.

Addresses of array elements.

array-addr	::=	array-addr-op.sval temp name vo index-list
array-addr-op	::=	AADDR | ATADDR

Temp must be a 32 bit quantity. Name is the name of a dimensioned variable (or a component in an array of records). Bp, vo, and index-list are defined as for an array load or store. Temp is assigned the address of the array element referenced. ATADDR is used when the variable is in a TEMPLATE packet; AADDR is used otherwise.

Note that name isn't needed to perform this operation. However, it is still provided to supply information to the optimizer.

Addresses of routines.

routine-addr	::=	RADDR temp name

Name is the name of an IL routine. Its address is assigned to temp, which must be a 32 bit quantity. This should never be used to implement a direct routine call, but rather should only be used when the address of a routine is explicitly taken.

Addresses of labels.

label-addr	::=	LADDR temp name

Name is a label name. Its address is assigned to temp, which must be a 32 bit quantity. This should not be used to implement anything other than the FORTRAN "ASSIGN" statement.

1.9 Routines

1.9.1 Definition

Routines begin with the ROUTINE pseudo-op. A routine header consists of loader information and a list of entry points.

routine-head	::=	ROUTINE loader-info { ENTRY entry-decl }

Loader-info is defined as for packets. Only DEFINED and INTERN are valid here. DEFINED means that the routine is defined here, but it may be called from another module, and the addresses of all its entry points must be supplied to the loader. INTERN means that the routine can only be called from other routines within the same module.

An entry declaration consists of the name of the entry, a label, and the name of the sole top-level record in a PARAM packet. Most routines will have only one entry. ⇐4.1

entry-decl	::=	entry-name result-type entry-label param-record
result-type	::=	32 | F32 | F64 | C64 | C128 | VOID
entry-label	::=	label-name

The entry-name identifies the entry. The name follows the IL rules for names. The CALL operator refers to a routine by an entry name.

Result-type indicates the type of the result returned by a call to this entry. VOID is used if no value is returned. F32 and F64 are used for those primitive data types. C64 and C128 are used for complex-valued routines. Otherwise, the returned value is an integer of some type or a pointer, in which case 32 is used.

The entry-label indicates where control is to be transferred when this entry is called. It labels a block of operations called the entry prologue. The entry prologue's successors are determined by the location of the entry prologue in the source program.

Param-record is the name of the sole top-level record in a PARAM packet. This record defines the format of the formal parameters, which are the top- ⇐4.1
level fields of the record. There is a distinct PARAM packet associated with each entry in a routine.

Consider a sample translation from FORTRAN. We assume call-by-reference. This means FORTRAN parameters map into IL pointers, and templates are introduced to describe the actuals referenced by the pointers. We assume a four cluster machine. (Translating FORTRAN is explained in the FORTRAN translation guide.)

```
SUBROUTINE SWAP(X,Y)
REAL X,Y
```

The translation of this FORTRAN declaration is a routine header with one entry point, a packet for the paramter block, and two template packets for the objects referenced by X and Y.

```
                        ; routine header
ROUTINE DEFINED                            -
    ENTRY SWAP VOID L1 1

                        ; packet for parameters
DCL 1 NIL OMOD32 8 PARAM INTERN            -
    VAR X 4 0 NIL NIL SCALAR P32           -
    VAR Y 4 4 NIL NIL SCALAR P32

                        ; template for actual referenced by X
DCL 2 NIL OMOD4 4 TEMPLATE INTERN          -
    VAR X' 4 0 NIL NIL SCALAR F32

                        ; template for actual referenced by Y
DCL 3 NIL OMOD4 4 TEMPLATE INTERN          -
    VAR Y' 4 0 NIL NIL SCALAR F32
```

1.9.2 Calls and returns

Calls. Routines are called with the CALL operator:

call-oper	::=	CALL.result-type result entry-name arg-name | CALLN.result-type result entry-name | PCALL.result-type result etemp arg-name | PCALLN.result-type result etemp
result	::=	temp | ctemp | NIL
etemp	::=	rvalue

⇐4.1

CALL and PCALL are used when actual arguments are supplied. CALLN and PCALLN are used when there are no actual arguments supplied. ⇐4.1

CALL and CALLN are used when the name of the entry is known; entry-name is the name. It must be declared in some ROUTINE-DECL. PCALL and PCALLN are used for indirect calls; etemp must be a pointer to the entry point. Result-type is the type of the value returned by this entry to the routine; if no value is returned, VOID is used.

The result temp is assigned the result returned by the routine. When result-type is VOID, result must be NIL. Ctemps are used for complex-valued routines (see definition of ctemp). ⇐4.1

Arg-name is the name of the RECORD to be used as the argument block. It must be the sole top-level variable in an ABLOCK packet.

Routines are exited with the RETURN operator:

return-oper	::=	RETURN.result-type ret-value
ret-value	::=	rvalue | crvalue | NIL

Result-type should match the declared type of the entry used on the invocation of the routine. (This correspondence cannot always be checked by the compiler for routines with multiple entries).

Ret-value is the value returned by the routine. No coercions are performed on the return value. Rvalue must be NIL if the result-type is VOID. Crvalues are used for complex-valued routines (see definition of crvalue).

To achieve the return of an aggregate object, the calling routine must allocate storage for the aggregate, and pass a pointer to it as an argument. The called routine must explicitly copy the aggregate value into the object referenced by the pointer.

1.10 Flow of Control

Flow of control within a routine is expressed with conditional and unconditional jumps. There are also specialized sets of looping instructions to be used in the translation of some source language looping constructs.

flow-oper	::=	label-oper
		| direct-jump
		| indirect-jump
		| conditional-jump
		| loop-control

1.10.1 Labels

Labels are defined with the LABEL pseudo-op.

label-oper	::=	LABEL label-name

Like any other IL name, each label name must be different from all other names in the module.

The target of any jump must be a label.

label	::=	label-name

When a jump is made to a defined label, control is transferred to the operation following the defining LABEL pseudo-op.

1.10.2 Direct jumps

There is one form of direct jump, a simple goto.

direct-jump	::=	goto-oper
goto-oper	::=	GOTO label

Control is transferred to the operation indicated by the label, as defined above. The target must be in the same routine as the goto operation.

Example. FORTRAN source.

```
      GOTO 100
      ...
100   CONTINUE
```

IL translation.

```
    GOTO L100
    ...
    LABEL L100
    ...
```

1.10.3 Indirect jumps

There is one form of indirect jump, an indirect goto.

indirect-jump	::=	igoto-oper
igoto-oper	::=	IGOTO temp { label-name }

Temp must contain a label address. A list of label names is optional. If the list is present, temp must contain the address of one of the labels in the list.

The address of a label may be taken with the LADDR operator, and it can be used in an indirect goto.

Example. FORTRAN source.

```
INTEGER I
...
ASSIGN 100 TO I
...
GOTO I,(100,200,300)
...
```

IL translation.

```
    ; assume I is in an ESTATIC packet
    ; store address of label in I
LADDR t1 L100
SSTORE.32 I t1
...
    ; load I and perform indirect goto
SLOAD.32 t2 I
IGOTO t2 L100 L200 L300
...
```

1.10.4 Conditional jumps

There are several forms of conditional jumps: jump if true or false, jump if a comparison is true or false, jump on the result of a generalized boolean ⇐4 operation, jump on the result of a generalized bitwise operation, and the switch operation.

conditional-jump	::=	if-true-oper
		| if-false-oper
		| if-comp-oper
		| if-boolean-oper
		| if-bitwise-oper
		| switch-oper

If-true and if-false operations test a temporary and jump to one of two targets based on the test. True is any non-zero value (when interpreted as a 32 bit integer); false is zero. These operations should never be used when some if-comp-oper can be used instead.

if-true-oper	::=	IF-TRUE rvalue probability label1 label2
if-false-oper	::=	IF-FALSE rvalue probability label1 label2
probability	::=	F32-constant | NIL

⇐4.1

Rvalue may be any non-F64 value. Control is transferred to label1 if the condition is met, label2 otherwise. The probability, if given, is the probability of branching to label1. If it is not known, NIL is used.

The if-comparison operations perform a comparison and jump to one of two targets based on the result.

if-comp-oper	::=	if-comp-op.if-comp-qualifier rvalue1 rvalue2 probability label-name1 label-name2
if-comp-op	::=	IF-EQ | IF-NE | IF-GT | IF-GE | IF-LT | IF-LE
if-comp-qualifier	::=	32 | 64 | I32 | SI32 | F32 | F64

The type of comparison performed is given by if-comp-qualifier. If if-comp-op is IF-EQ or IF-NE, then if-comp-qualifier must be 32 or 64. If if-comp-op is IF-GT, IF-GE, IF-LT, or IF-LE, then if-comp-qualifier must be I32, SI32, F32, or F64. The only restrictions on the types of rvalue1 and rvalue2 are:

- If if-comp-qualifier is 32, I32, SI32, or F32, then neither rvalue1 nor rvalue2 may have type F64.
- If if-comp-qualifier is 64 or F64, then both rvalue1 and rvalue2 must have type F64.

The indicated comparison operation is performed on rvalue1 and rvalue2. No coercions are perfromed. If the result is true, control is transferred to label-name1; otherwise control is transferred to label-name2. Probability is as defined above.

Example. FORTRAN source.

```
INTEGER A,B
...
IF (A.GT.B) GOTO 100
...
```

IL translation.

```
  ; assume A and B are in ESTATIC packets
  ; load A and B
```

```
LOAD.32 t1 A
LOAD.32 t2 B
    ;
    ; conditionally branch on A > B
IF-GT.SI32 t1 t2 NIL L100 L1
LABEL L1
...
```

The generalized boolean conditionals perform a generalized boolean operation and jump to one of two targets based on the result.

if-boolean-oper	::=	if-nbop { (NOT | NIL) rvalue } probability label-name1 label-name2
if-nbop	::=	IF-GAND | IF-GOR | IF-GEQV | IF-GNEQV

⇐4

The if-nbop operations correspond to the generalized boolean operations defined in a later section; look ahead for details.

Example. FORTRAN source.

```
      LOGICAL A,B
      ...
      IF (A.EQV.B) GOTO 100
      ...
```

IL translation.

```
    ; load A and B
LOAD.32 t1 A
LOAD.32 t2 B
    ;
    ; conditionally branch on A.EQV.B
IF-GEQV t1 t2 NIL L100 L1
LABEL L1
...
```

The generalized bitwise conditionals perform a generalized bitwise operation and jump to one of two targets based on the result.

if-bitwise-oper	::=	if-nbtop { (BNOT | NIL) rvalue } probability label-name1 label-name2
if-nbtop	::=	IF-GBAND | IF-GBOR | IF-GBEQV | IF-GBNEQV

⇐4

The if-nbtop operations correspond to the generalized bitwise operations defined in a later section; look ahead for details. The result of a bitwise operation is true if it is non-zero.

Example. C source.

```
unsigned int a, b;
...
if (a & b) goto l100
...
```

IL translation.

```
    ; load a and b
LOAD.32 t1 a
LOAD.32 t2 b
    ;
    ; conditionally branch on a & b
IF-GBAND t1 t2 NIL L100 L1
LABEL L1
...
```

A switch operation permits a selection from a sequence of targets. ⇐4

switch-oper	::=	SWITCH rvalue probability label { 32-bit-constant probability label }

⇐4.1

The operands of a SWITCH operation are an rvalue, a probability and label pair, and a list of constant, probability, and label triples.

Each constant is 32 bits. The label triples must be sorted in ascending order ⇐4.1
of the constants, where each constant is interpreted as a signed integer for the purposes of this ordering (no coercions are performed). Each constant in the list must be unique (when interpreted as signed integers); labels may be repeated.

The rvalue may have any primitive data type other than F64.

The list of triples is searched for a constant equal to the value of temp. A 32 bit comparison is used, with no coercions performed. If a match is found, control is transferred to the label paired with the constant. If no match is found, control is transferred to the first (default) label. However, if the first probability and label are both NIL, then it is assumed that rvalue will always match one of the constants, and no dynamic range checking will be performed.

The probability, if given, defines the probability of selecting the alternative. Either all the probabilities should be defined in a switch, of all should be NIL. The probabilities should sum to one.

Example. FORTRAN source.

```
      INTEGER I
      ...
      GOTO (100,200,300),I
```

IL translation.

```
        ; assume I is in an ESTATIC packet
        ; load I
    LOAD.32 t1 I
        ;
        ; if t1 is one of 1,2,3 control goes to
        ; corresponding one of L100,L200,L300
        ; else control goes to default label
    SWITCH t1 0.25 L1 1 0.25 L100 2 0.25 L200 3 0.25 L300
    LABEL L1
```

1.10.5 Loop control

There are two pairs of loop control operations. A pair of pseudo-operations is used by Phase 1 to delimit source-level loop constructs. A second pair of operations is used to translate the FORTRAN DO-loop.

loop-control	::=	loop-begin-oper
		| loop-end-oper
		| do-begin-oper
		| do-end-oper

The LOOP-BEGIN, LOOP-END pair of pseudo operations are used to delimit loops formed from WHILE, REPEAT and other high level loop constructs. They are not used if a (DO-BEGIN, DO-END) pair already delimit the loop. The purpose of the loop pseudo ops is to aid the control-flow analyzer in determining the loop structure of the program.

loop-begin-oper	::=	LOOP-BEGIN loop-number expected-iterations
loop-end-oper	::=	LOOP-END loop-number
loop-number	::=	I32-constant
expected-iterations	::=	I32-constant | NIL

The loop number identifies the loop and permits us to pair loop-begins with loop-ends easily. Loop-begins and loop-ends may be nested. If the expected number of iterations of the loop is known at compile time, it is given; otherwise NIL is used.

A loop-end should be interpreted as a direct jump to the corresponding loop-begin.

There are two operations to support FORTRAN DO-loops.

do-begin-oper	::=	DO-BEGIN.do-type loop-number expected-iterations do-var init final incr
do-end-oper	::=	DO-END loop-number
do-type	::=	SI32 | F32 | F64
do-var	::=	name (bp | NIL)
init	::=	rvalue
final	::=	rvalue
incr	::=	rvalue

For each do-begin there must be a matching do-end; they can easily be paired using the loop-number.

The expected iterations are given if an estimate is known at compile time; otherwise NIL is used.

Dovar consists of an IL variable name (the DO-variable) and an optional base pointer. The base pointer must be present if it is needed to address the packet containing the DO-variable. The DO-variable must have the same type as do-type.

Each of init, final, and incr is an rvalue. The only restrictions on the types of init, final, and incr are:

- If do-type is SI32 or F32, then neither init, final, nor incr may have type F64.
- If do-type is F64, then init, final, and incr must all have type F64.

Note that no coercions are performed; when there is a type mismatch, the value is interpreted as if it had the type of do-var. Therefore, the front-ends must explicitly coerce operands of a different type when it is required.

The body of the loop will be determined graphically. Loop control is defined as for FORTRAN. The loop will be executed `MAX( n, INT( (final - init + incr ) / incr ) )` times, where n is 0 for F77 and 1 for F66 (and arithmetic is performed as indicated by do-type, with no coercions performed). The DO-variable will be initialized to init and incremented each time the do-end is reached.

This can be expanded into several possible sequences of operations, depending on the values of init, final, and incr. It is not a direct transliteration of the FORTRAN source, as init, final, and incr are all rvalues, not expressions.

Example. FORTRAN source.

```
      INTEGER I
      ...
      DO 100 I = 1, 10
      ...
100   CONTINUE
      ...
```

IL translation.

```
        ; assume I is in an ESTATIC packet
        ; the label 100 must be kept in the translated body,
        ; unless it is only referenced by DO statements
    DO-BEGIN.SI32 22 10 I NIL 1 10 1
    ...
    LABEL L100
    DO-END 22
    ...
```

1.11 Assignments and type coercions

This section presents the assigning operations, which permit 32 and 64 bit values to be copied from one temporary to another with no change, and the coercion operations, which permit 32 and 64 bit values to be copied from one temporary to another, with coercion performed on the value.

1.11.1 Assignments

assign-oper	::=	ASSIGN.assign-size temp rvalue
assign-size	::=	32 | 64

The ASSIGN operator copies an rvalue to a temp. The only restrictions on the types of temp and rvalue are:

- If assign-size is 32, then neither temp nor rvalue may have type F64.
- If assign-size is 64, then both temp and rvalue must have type F64.

1.11.2 Coercions

coercion-oper	::=	CONV.temp-data-type.prim-data-type temp rvalue

A coercion operation coerces rvalue to temp. Rvalue is interpreted as being of type prim-data-type, and temp is interpreted as being of type temp-data-type. The only coercion performed is from the prim-data-type to the temp-data-type. The only restrictions on the types of temp and rvalue are:

- If prim-data-type is not F64, then rvalue must not have type F64.
- If prim-data-type is F64, then rvalue must have type F64.
- If temp-data-type is not F64, then temp must not have type F64.
- If temp-data-type is F64, then temp must have type F64.

The result of any coercion operation can have a different representation from its source operand (i.e., there are no ”do-nothing” coercion operators).

There has been some confusion as to what it means to convert an integer of one type to a smaller integer of another type. We define this as follows. First truncate the 32 bit source operand to the number of bits indicated by the result type. This is done by eliminating the leftmost excess bits. Then either zero-fill the left part of the result (for conversions to some In) or sign-extend the left part of the result from the leftmost bit of the truncated source operand (for conversions to some SIn). The result is a 32 bit quantity whose rightmost n bits match the rightmost n bits of the original source operand.

Below is a table of the valid combinations of temp-data-type and prim-data-type. Type1 is the source type (i.e., prim-data-type), and type2 is the destination type (i.e., temp-data-type). Note that this does not place any restrictions on the types of rvalue and temp, except as noted above. This table merely indicates the IL coercion operators that are available. An "X" indicates a legal combination; a "-" indicates an illegal combination.

	type2							
type1	I1 ... I31	I32	SI8	SI16	SI32	P32	F32	F64
I8	-	-	-	-	-	-	-	-
I16	-	-	-	-	-	-	-	-
I32	X	-	-	-	-	-	X?	X?
SI8	-	-	-	-	-	-	-	-
SI16	-	-	-	-	-	-	-	-
SI32	-	-	X	X	-	-	X	X
P32	-	-	-	-	-	-	-	-
F32	-	X?	-	-	X	-	-	X
F64	-	X?	-	-	X	-	X	-

1.12 Scalar operations

The scalar operations all have a similar structure. A function is applied to the source rvalue(s), and the result assigned to the destination temp(s). Coercion is never performed on any of the operands; in fact, the only restriction on the types of the operands is that they be of the appropriate size for the operation (i.e., 32 vs. 64 bits). Small integers are treated as 32 bit quantities. In addition, certain arithmetic operations are defined for complex types (C64 and C128). These operations take pairs of operands, where the first member of each pair is taken to be the real part, and the second member is taken to be the imaginary part. These complex components are interpreted as F32 or F64 values, although once again the only restriction on the types is that they be of the correct size. No coercions are performed. These complex operations either be expanded in-line or replaced with function calls during Phase 2.

There are several categories of scalar operations.

scalar-oper	::=	arithmetic-oper
		| boolean-oper
		| relational-oper
		| bit-oper
		| select-oper

We present each in turn.

1.12.1 Arithmetic operations

Arithmetic operations are either unary, binary, or n-ary.

arithmetic-oper	::=	unary-arithmetic-oper
		| binary-arithmetic-oper
		| nary-arithmetic-oper

Unary. The unary arithmetic operations are negation and reciprocal.

unary-arithmetic-oper	::=	unaop.unaop-type dest source
unaop	::=	NEG | RECIP
unaop-type	::=	I32 | SI32 | F32 | F64 | C64 | C128
dest	::=	temp | ctemp
source	::=	rvalue | crvalue

In all cases, dest is assigned the result of unop source.

For NEG, rvalue is simply negated.

For RECIP, the reciprocal of rvalue is taken. This operation is somewhat bizarre when applied to integers, but is nevertheless defined here for completeness.

Dest and source are ctemp and crvalue only when unaop-type is C64 or C128. The only type restrictions are that all operands must be of the correct size. No coercions are performed.

Binary. The binary arithmetic operations are power, and integer division, remainder and mod. The last three are grouped together as integer division class operators.

binary-arithmetic-oper	::=	power-oper | int-div-class-oper

Power operator. The binary power operator raises one number to the power of another number.

power-oper	::=	POWER.base-type.exp-type dest base exp
base-type	::=	SI32 | F32 | F64 | C64 | C128
exp-type	::=	SI32 | F32 | F64 | C64 | C128
dest	::=	temp | ctemp
base	::=	rvalue | crvalue
exp	::=	rvalue | crvalue

Base is raised to the exp power, and the result is stored in dest. Dest and base are interpreted as having type base-type, and exp is interpreted as having type exp-type. Base-type and exp-type must be the same, unless exp-type is SI32, in which case base-type may be any of the above possibilities.

Dest and base are ctemp and crvalue only when base-type is C64 or C128. Exp is crvalue only when exp-type is C64 or C128. The only type restrictions are that all operands must be of the correct size. No coercions are performed.

Integer division class operators. The binary integer division class operators are division, remainder, and mod.

int-div-class-oper	::=	divcop.divcop-type temp rvalue1 rvalue2
divcop	::=	DIV | REM | MOD
divcop-type	::=	I32 | SI32

In all cases, temp is assigned the result of rvalue1 divcop rvalue2. Temp, rvalue1, and rvalue2 must all be 32 bit values. They are interpreted as being of type divcop-type. No coercion is performed.

In all cases, rvalue2 must not be zero when interpreted as a divcop-type quantity.

DIV truncates toward zero (e.g. (-8)/3 = -2).

REM is the remainder for integer division, i.e., the remainder of a and b is given by `REM(a,b) = a - int(a/b)*b`, where `int` rounds towards zero.

The remainder has the same sign as a. (This is the C % operator and the Fortran intrinsic function MOD (even though it's a stupid definition for MOD).)

MOD is defined as `MOD(a,b) = a - floor(a/b)*b`, where `floor` rounds down. MOD is identical to REM if both operands are positive. (This is the PASCAL MOD operator.)

Note that `REM(a,b) = MOD(a,sign(a)*abs(b))`.

The following table contrasts the IL operators with those of FORTRAN, C, and PASCAL.

operands		IL			FORTRAN		C		PASCAL	
a	b	IDIV	REM	MOD	/	MOD	/	%	DIV	MOD
5	3	1	2	2	1	2	1	2	1	2
5	-3	-1	2	-1	-1	2	-1	2	-1	error
-5	3	-1	-2	1	-1	-2	-1	-2	-1	1
-5	-3	1	-2	-2	1	-2	1	-2	1	error

(Note: FORTRAN and C don't require the IL MOD operator. We are not supporting it for now.)

N-ary. There are two n-ary operators. The first, called GADD, combines addition and subtraction. The second, called GMUL, combines multiplication and floating division. An n-ary operation defines an expression which can be arbitrarily reassociated and commuted when translated into binary and unary RISC-level operations. If a source-level expression does not have this property, it should be translated into a sequence of subexpressions which do.

nary-arithmetic-oper	::=	nary-add-oper | nary-mul-oper
nary-add-oper	::=	GADD.nary-arith-type dest { (NEG | NIL) source }
nary-mul-oper	::=	GMUL.nary-arith-type dest { (RECIP | NIL) source }
nary-arith-type	::=	I32 | SI32 | F32 | F64 | C64 | C128
dest	::=	temp | ctemp
source	::=	rvalue | crvalue

There must be at least two sources in the operand list. Dest is assigned the sum or product of the sources, with the following exceptions:

- If a source in a GADD operation is preceded by a NEG, its arithmetic negation should be used.
- If a source in a GMUL operation is preceded by a RECIP, its reciprocal should be used. RECIP cannot be used when nary-arith-type is I32 or SI32.

No ordering of the individual additions or multiplications within a given n-ary operation is defined.

Dest and each source are ctemp and crvalue only when nary-arith-type is C64 or C128. The only type restrictions are that all operands must be of the correct size. No coercions are performed.

(Note: Add restrictions on pointers.)

1.12.2 Boolean operations

Boolean operations are either unary or n-ary.

boolean-oper	::=	unary-boolean-oper
		| nary-boolean-oper

Boolean operation operate on 32 bit quantities and produce a 32 bit result. When interpreted as a (signed or unsigned) integer, the result is either 1 (true) or 0 (false). "Unnormalized" boolean values are accepted as input; a non-zero value is interpreted as true, and zero is false.

Unary. The unary boolean operations are NORM and NOT.

unary-boolean-oper	::=	unbop temp rvalue
unbop	::=	NORM | NOT

NORM is 1 if rvalue is non-zero, 0 otherwise. NOT is 1 if rvalue is zero, 0 otherwise.

The only type restrictions are that all operands must be of the correct size. No coercions are performed.

N-ary. The n-ary boolean operations are GAND, GOR, GEQV, and GNEQV.

nary-boolean-oper	::=	nbop temp { (NOT | NIL) rvalue }
nbop	::=	GAND | GOR | GEQV | GNEQV

There must be at least two rvalues in the operand list. Temp is assigned the boolean result of applying nbop to the rvalues, with the following exception:

- If an rvalue is preceded by a NOT, its logical negation should be used.

No ordering of the individual boolean ANDs, ORs, EQVs, or NEQVs within a given n-ary operation is defined.

We follow the FORTRAN definitions of EQV and NEQV, i.e., NEQV is the same as ”exclusive or”, and EQV is the same as ”exclusive nor”.

The only type restrictions are that all operands must be of the correct size. No coercions are performed.

1.12.3 Relational operations.

The relational operations are all binary.

relational-oper	::=	relbop.comp-qualifier temp rvalue1 rvalue2
relbop	::=	EQ | NE | GT | GE | LT | LE
comp-qualifier	::=	32 | 64 | I32 | SI32 | F32 | F64

The indicated comparison operation is performed on rvalue1 and rvalue2, and the logical result is stored in temp. Temp is interpreted as a (signed or unsigned) integer; its value is either 1 (true) or 0 (false).

The type of comparison performed is given by comp-qualifier. If relbop is EQ or NE, then comp-qualifier must be 32 or 64. If relbop is GT, GE, LT, or LE, then comp-qualifier must be I32, SI32, F32, or F64.

The only type restrictions are that all operands must be of the correct size. No coercions are performed.

1.12.4 Bit operations

Bit operations are either unary, binary, or n-ary.

bit-oper	::=	unary-bit-oper
		| binary-bit-oper
		| nary-bit-oper

Boolean operation operate on 32 bit quantities and produce a 32 bit result.

Unary. The unary bit operator is BNOT.

unary-bit-oper	::=	unbtop temp rvalue
unbtop	::=	BNOT

Rvalue is bitwise complemented, and the result is stored in temp.

The only type restrictions are that all operands must be of the correct size. No coercions are performed.

Binary. The binary bit operators are LSL, LSR, ASHL, and ASHR.

binary-bit-oper	::=	binbtop temp rvalue1 rvalue2
binbtop	::=	LSL | LSR | ASHL | ASHR

Rvalue1 is shifted by the number given in rvalue2, and the result stored in temp. LSL is a logical left shift, LSR is a logical right shift, ASHL an arithmetic left shift, and ASHR an arithemtic right shift. Rvalue2 is ⇐4.1
interpreted as an I32 value, and must be < 32 (i.e., $0 \leq$ rvalue2 ≤ 31).

The only type restrictions are that all operands must be of the correct size. No coercions are performed.

N-ary. The n-ary bit operators are GBAND, GBOR, GBEQV, and GBNEQV.

nary-bit-oper	::=	nbtop temp { (BNOT | NIL) rvalue }
nbtop	::=	GBAND | GBOR | GBEQV | GBNEQV

There must be at least two rvalues in the operand list. A bitwise AND, OR, EQV, or NEQV is performed on the rvalues, and the result stored in temp, with the following exception:

- If an rvalue is preceded by a BNOT, its bitwise inverse should be used.

The only type restrictions are that all operands must be of the correct size. No coercions are performed.

1.12.5 Select operations

For each conditional jump (except for IF-TRUE, IF-FALSE, and SWITCH) there is a corresponding select operation. In a select operation, a conditional is evaluated to select one of two values. The selected value is assigned to the result temp.

The comparison selects:

<table>
<tr><td>slct-comp-oper</td><td>::=</td><td>slct-comp-op.slct-comp-qualifier.slct-val-qualifier
sresult rvalue1 rvalue2 tvalue fvalue</td></tr>
<tr><td>slct-comp-op</td><td>::=</td><td>SLCT-EQ | SLCT-NE | SLCT-GT | SLCT-GE | SLCT-LT | SLCT-LE</td></tr>
<tr><td>slct-comp-qualifier</td><td>::=</td><td>32 | 64 | I32 | SI32 | F32 | F64</td></tr>
<tr><td>slct-val-qualifier</td><td>::=</td><td>32 | 64 | C64 | C128</td></tr>
<tr><td>sresult</td><td>::=</td><td>temp | ctemp</td></tr>
<tr><td>tvalue</td><td>::=</td><td>rvalue | crvalue</td></tr>
<tr><td>fvalue</td><td>::=</td><td>rvalue | crvalue</td></tr>
</table>

$\Leftarrow$4

Rvalue1 and rvalue2 are compared. If the result is true, tvalue is assigned $\Leftarrow$4.1
to sresult, else fvalue is.

Slct-val-qualifier gives the type (or size) of tvalue, fvalue, and sresult. Both values must be of size 32 or less, or both must be of type F64, or both of C64, or both of C128.

The type of comparison performed is given by slct-comp-qualifier. If slct-comp-op is SLCT–EQ or SLCT–NE, then slct-comp-qualifier must be 32 or 64. If slct-comp-op is SLCT–GT, SLCT–GE, SLCT–LT, or SLCT–LE, then slct-comp-qualifier must be I32, SI32, F32, or F64. The only restrictions on the types of rvalue1 and rvalue2 are:

- If slct-comp-qualifier is 32, I32, SI32, or F32, then neither rvalue1 nor rvalue2 may have type F64.
- If slct-comp-qualifier is 64 or F64, then both rvalue1 and rvalue2 must have type F64.

The generalized boolean selects:

slct-boolean-oper	::=	slct-boolean-op.slct-val-qualifier sresult { (NOT | NIL) rvalue } tvalue fvalue
slct-boolean-op	::=	SLCT-GAND | SLCT-GOR | SLCT-GEQV | SLCT-GNEQV

⇐4

The generalized boolean selects are defined like the comparison selects, except that a generalized boolean operation is used as a test.

The generalized bitwise selects:

slct-bitwise-oper	::=	slct-bitwise-op.slct-val-qualifier sresult { (BNOT | NIL) rvalue } tvalue fvalue
slct-bitwise-op	::=	SLCT-GBAND | SLCT-GBOR | SLCT-GBEQV | SLCT-GBNEQV

⇐4

The generalized bitwise selects are defined like the comparison selects, except that a generalized bitwise operation is used as a test. The result of a bitwise operation is true if the result is non-zero.

1.13 Assertion operations

Assertion operations are intended to supply information to the disambiguator to aid it in proving facts about memory references. Assertion operations are predicates which must be true every time they are evaluated. They are removed from the program by Phase 2. We might eventually wish to be able to leave them in the program, in which case they would give an error message whenever they failed. ⇐4.1

Assertions are useful for supplying information to the disambiguator which is either too obscure, tricky, or complicated for it to figure out for itself, or which it cannot correctly assume without being specifically informed (e.g., one might assert something about a formal routine argument, which would place restrictions on how the routine could be called but could result in better code). ⇐4.1

There are several types of assertion operations. ⇐4.1

assert-oper	::=	assert-true-oper | assert-false-oper | assert-comp-oper

We present each in turn.

(Note: We may eventually wish to add assertions which could be used to indicate the alignment of formals.)

1.13.1 True/false assertion operations.

These are unary operations.

assert-true-oper	::=	ASSERT-TRUE rvalue
assert-false-oper	::=	ASSERT-FALSE rvalue

Rvalue may be any non-F64 value. True is any non-zero value (when interpreted as a 32 bit integer); false is zero. These operations should never be used when some assert-comp-oper can be used instead. ⇐4.1

These operations assert that the condition is always met when the assertion is executed.

1.13.2 Comparison assertion operations.

The comparison assertion operations are all binary.

assert-comp-oper	::=	acomop.comp-qualifier rvalue1 rvalue2
acomop	::=	ASSERT-EQ | ASSERT-NE | ASSERT-GT | ASSERT-GE | ASSERT-LT | ASSERT-LE
comp-qualifier	::=	32 | 64 | I32 | SI32 | F32 | F64

The operations assert that the indicated comparison, if performed on rvalue1 and rvalue2, must be true. ⇐4.1

The type of comparison performed is given by comp-qualifier. If acomop is ASSERT-EQ or ASSERT-NE, then comp-qualifier must be 32 or 64. If acomop is ASSERT-GT, ASSERT-GE, ASSERT-LT, or ASSERT-LE, then comp-qualifier must be I32, SI32, F32, or F64.

The only type restrictions are that all operands must be of the correct size. No coercions are performed.

1.14 Special Operations

There are a number of special operations which are provided to support various high-level operations such as character concatenation, structure copy, etc. Phase 2 will either expand these operations in-line or convert them into calls to special routines.

There are several categories of special operations.

special-oper	::=	char-string-oper
		| record-copy-oper
		| array-copy-oper

1.14.1 Character string operations

There are several character string operations required to support F77 character operations.

char-string-oper	::=	char-copy-oper
		| char-concat-oper
		| char-compare-oper

Each character string operation uses one or more character descriptors as operands. These are merely pairs of rvalues; the first is a pointer to a character string and the second is the number of characters in the string.

char-desc	::=	char-string-ptr char-string-len
char-string-ptr	::=	rvalue
char-string-len	::=	rvalue

In all cases, char-string-ptr is interpreted as a P32 value which points to the first character of the character string, and char-string-len is interpreted as a 32 bit unsigned integer. Char-string-len must not be zero.

The only type restrictions on char-string-ptr and char-string-len is that they be 32 bit quantities. No coercions are performed.

Character string copy The character string copy operator is CSCOPY.

char-copy-oper	::=	CSCOPY char-desc1 char-desc2

The character string decribed by char-desc2 is copied into the character string described by char-desc1. If the source string is longer than the destination string, then the rest of the source string is ignored. If the source string is shorter than the destination string, then the rest of the destination string is blank-filled.

Character string concatenation The character string concatenation operator is CSCONCAT.

char-concat-oper	::=	CSCONCAT dest-desc { source-desc }
dest-desc	::=	char-desc
source-desc	::=	char-desc

There must be at least two source-descs in the operand list. The character strings decribed by the source-descs are concatenated (left-to-right) into the character string described by dest-desc. Copying stops when the destination string is full. If there aren't enough characters to fill the destination string, then the rest of the destination string is blank-filled.

Character string comparison The character string comparison operators are CSEQ, CSNE, CSGT, CSGE, CSLT, and CSLE.

char-compare-oper	::=	cscompop temp char-desc1 char-desc2
cscompop	::=	CSEQ | CSNE | CSGT | CSGE | CSLT | CSLE

The indicated comparison is performed on the two character strings, and the result is stored in temp. Temp is interpreted as a (signed or unsigned) integer; its value is either 1 (true) or 0 (false). It must not have type F64. If one of the character strings is shorter than the other, it is treated as if it were padded out to the longer length with blanks.

1.14.2 Record copy operations

There are currently four record copy operators. ⇐4.1

record-copy-oper	::=	RCOPY.N.N d-name s-name words | RCOPY.N.T d-name s-name s-bp words | RCOPY.T.N d-name d-bp s-name words | RCOPY.T.T d-name d-bp s-name s-bp words
words	::=	I32-constant

⇐4.1

D-name and S-name are the names of the destination and source records. D-bp and s-bp are the source and destination base pointers. They are present when the corresponding name is in a template packet. ⇐4.1

RCOPY.N.N is used for non-template to non-template copies. RCOPY.N.T is used for template to non-template copies. RCOPY.T.N is used for non-template to template copies. RCOPY.T.T is used for template to template copies. ⇐4.1

Words is a constant indicating how many words are to be copied. ⇐4.1

The only type restrictions are that all operands must be of the correct size. No coercions are performed.

1.14.3 Array copy operations

There are currently four array copy operators. ⇐4.1

array-copy-oper	::=	ACOPY.N.N d-name s-name rvalue
		| ACOPY.N.T d-name s-name s-bp rvalue
		| ACOPY.T.N d-name d-bp s-name rvalue
		| ACOPY.T.T d-name d-bp s-name s-bp rvalue

D-name and S-name are the names of the destination and source arrays. D-bp and s-bp are the source and destination base pointers. They are present when the corresponding name is in a template packet. ⇐4.1

ACOPY.N.N is used for non-template to non-template copies. ACOPY.N.T is used for template to non-template copies. ACOPY.T.N is used for non-template to template copies. ACOPY.T.T is used for template to template copies. ⇐4.1

Rvalue is interpreted as an I32 value which indicates the number of bytes(?) to copy. ⇐4.1

The only type restrictions are that all operands must be of the correct size. No coercions are performed.

(Note: These operations are currently NOT supported.) ⇐4.1

Chapter 2

IL-2 (low-level subset)

2.1 Basics

IL-2 defines a portion of the Phase 2-3 interface. At the interface, individual IL-2 operations will be expressed by IL data abstractions. Flow of control will be expressed a flow graph. A full definition of the interface depends on these data structures as well as IL-2. The concrete syntax of IL-2 is not important except for listings, etc, and in this chapter, we describe each operation as an op-code and a list of operands.

The IL-2 operations correspond closely to machine operations. The operands are either temporaries or constants (rvalues) or special 'disambiguator operands' (see discussion of loads and stores below).

All information the code generator needs about the operation is available in the operation; the code generator should not need access to the Phase 2 symbol table, except to add new variables (for spilling) and to determine how much automatic storage is required by a routine. Aside from this, memory allocation will be performed in Phase 2. The trace scheduler will also need live variable information, though this can be captured once at the beginning of Phase 3.

Some IL-2 operations have the same name as machine operations; others have the same name as IL-1 operations; others are unique. The names are meant to specify IL-2 operations, and when the context is not clear, they should be prefixed with "IL-2.".

2.2 Names

Since the Phase 2/Phase 3 interface is not purely textual, declared names ⇐4.1
are available in a pre-constructed symbol table. Names may be examined or added according to the symbol table interface.

The names themselves have the same format as IL-1 names (see the IL-1 description), with the following exception:

- A name may now contain the two-character sequence consisting of a question mark "?" followed by the digit two "2", i.e., the sequence "?2".

Names containing "?2" will have been introduced during Phase 2. As was ⇐4.1
indicated in the IL-1 document, the idea is that each phase of the compiler will be free to introduce names with the sequence "?n", where n is the phase number, without fear of conflicting with a name created before that phase.

Phase 3 is therefore free to create names containing the sequence "?3". At the end of Phase 3, all question marks "?" appearing in names must be immediately followed by one of the digits "1", "2", or "3".

2.3 Constants

Constants in the IL-2 are untyped bit strings. Allocation is performed in Phase 2, and all addressing constants are resolved to a known value or to a load-time expression. Thus IL-2 will contain two types of constants: literal constants whose value and size in bits is known, and symbolic constants whose value will be determined at load time, and whose size is assumed to be 32 bits.

IL-2 permits constants to be used as immediate operands wherever the hardware does. There are several cases to consider:

- 32 bit (or shorter) constants used in non-floating operations must always be on the right ('B') leg of the operation.
- 32 bit (or shorter) constants used in floating operations must be explicitly assigned to temporaries. (It is our understanding that the floating operations cannot access the immediate fields of an instruction.)

- 64 bit temporaries containing constants must be explicitly constructed by setting the left and right half, using the assignment operations defined below.

2.4 Assignments

There are five IL-2 assignment operations.

ASS.I32	temp rvalue
ASS.F32	temp rvalue
ASSL.64	temp rvalue
ASSR.64	temp rvalue
ASS.64	temp temp

ASS.I32 and ASS.F32 assign the right operand to a new 32 bit temporary. The .I32 and .F32 are hints to the code generator to indicate the result would prefer to be in an integer or floating bank, but the code generator is free to ignore the hint.

ASSL.64 and ASSR.64 assign the left and right halves of a 64 bit floating point temporary. The unassigned half is not changed, and it is undefined if the temporary has not been previously defined. These operations are to be used to construct 64 bit values from two 32 bit constants.

ASS.64 copies values between 64 bit temporaries. Note it cannot have a constant as a right operand. ⇐4.1

(Note: If the hardware provides a mechanism for moving an F64 register into two integer registers, then we will add appropriate IL-2 operators which can be introduced during Phase 2 when appropriate.) ⇐4.1

(Note: We will resolve the loading of F32 constants into F32 registers when the hardware is better understood.)

2.5 Loads and Stores

All IL-1 memory references will be fully expanded in IL-2 to machine level address arithmetic, loads, stores, and where necessary extracts, merges, shifts and masks. The operands to the IL-2 load and store operations will be temporaries and constants, as desired by the code generator. There is an additional operand identifying the IL-1 packet or variable being loaded and stored, for use by the disambiguator. TJK must specify what he wants this operand to be. The IL abstractions will be defined so that the code generator, the disambiguator, and Phase 2 can each view the operands in the way that they desire.

All load operations have the form

ld-op temp1 temp2 rvalue disamb-operand

There are seven IL-2 operations to load a 32 bit value from memory, and two operations to load a 64 bit value:

32 bit load	64 bit load
LD.8	LD.64
LD.16	SLD.64
SLD.16	
LD.I32	
LD.F32	
SLD.I32	
SLD.F32	

In all of the operations, temp1 is assigned the loaded value, temp2 and rvalue are used to calculate the address, and disamb-operand is information needed by the disambiguator. In a plain LD, the address is $temp2 + rvalue$. In an SLD, the address is $scale \times temp2 + rvalue$. The operation qualifiers determine the scale, and they also determine when to trap on a misaligned read.

qualifier	scale	trap condition
.8	-	never
.16	2	MOD(address,2) $\neq 0$
.I32,.F32	4	MOD(address,4) $\neq 0$
.64	8	MOD(address,8) $\neq 0$

The .I32 and .F32 are hints to the code generator to indicate the result would prefer to be in an integer or floating bank, but the code generator is free to ignore the hint. (Code generator warning: The hardware instructions LD.I32 and LD.F32 are different instructions, with different resource requirements.)

All store operations have the form

st-op temp1 temp2 rvalue disamb-operand

There are five IL-2 operations to store a 32 bit value to memory, and two operations to store a 64 bit value.

32 bit store	64 bit store
ST.8	ST.64
ST.16	SST.64
SST.16	
ST.32	
SST.32	

In all of the operations, temp1 is the value to be stored, temp2 and rvalue are used to calculate the address, and disamb-operand is information needed by the disambiguator. In a plain ST, the address is $temp2 + rvalue$. In an SST, the address is $scale \times temp2 + rvalue$. The operation qualifiers determine the scale, and they also determine when to trap on a misaligned store.

qualifier	scale	trap condition
.8	-	never
.16	2	MOD(address,2) $\neq$ 0
.32	4	MOD(address,4) $\neq$ 0
.64	8	MOD(address,8) $\neq$ 0

Some examples (check these):

- Scalar loads and stores:

```
IL-1:

    DCL 1 NIL OMOD4 4 AUTO INTERN
        VAR I 4 0 NIL NIL SCALAR SI32

    SLOAD.32 T1 I


IL-2:

    LD.I32 T1 SP offset_I packet_1
```

```
; SP is the stack pointer in the current routine
; offset_I is the byte offset of I from SP
;          = offset_packet1 + offset_I_in_Packet
; packet_1 is the operand for the disambiguator
```

- Byte and half-word loads and stores:

```
IL-1:

    DCL 1 NIL OMOD4 1 AUTO INTERN
        VAR C 1 0 NIL NIL SCALAR I8

    SLOAD.I8 T5 C
    ...
    SSTORE.I8 C T16

IL-2:

    LD.8 T1 SP offset_C packet_1       ; load 32-bit word containing C
    ADD.U T2 SP offset_C               ; recalculate address
    EXT.UI8 T5 T1 T2                   ; extract appropriate byte.

    ....

    LD.8 T7 SP offset_C packet_1       ; load 32-bit word containing C
    ADD.U T8 SP offset_C               ; recalculate address
    MRG.UI8 T9 T7 T8 T16               ; merge in byte from T16
    ST.8 T9 SP offset_C packet_1       ; store result
```

- Field loads and stores: The bit numbering in field loads and stores is left to right as defined IL-1.

```
C:
    static struct {
        unsigned is-keyword : 1;
        unsigned is-extern  : 1;
        unsigned is-static  : 1;
    } flags;
```

```
    x = flags.is_extern;


IL-1:
                            ; packet for flags
    DCL 1 NIL OMOD4 4 STATIC INTERN              -
        VAR flags 4 0 NIL NIL                    -
            RECORD OMOD4                         -
                COMPONENT xflags 4 0 NIL NIL     -
                    PACKED                       -
                        FIELD is-keyword 0 1     -
                        FIELD is-extern  1 1     -
                        FIELD is-static  2 1     -
                        END-PACKED               -
                END-RECORD


    FLOAD.1 T5 is-extern


IL-2:

    LD.I32 T1 AP offset_is-extern packet_1    ; load 32 bit word containing
                                              ;   is-extern
    LSR T2 T1 30                              ; shift bit to right
    GBAND T5 T2 1                             ; bitwise AND with 1
```

2.6 Entries, Calls, and Returns

The entry, call, and return operations are responsible for performing the procedure call mechanism. The tasks are to pass the return address and the argument block pointer to a called routine, to pass a return value back from a called routine, and to keep track of the stack pointer at all times. There is no activation pointer, as all local references will be made from the stack pointer. (Note: We may also have to maintain a pointer into the spill bank across calls.)

The stack will be used in a stylized manner: it will grow down in memory; all of the local storage required by a routine will be allocated once at the beginning of the routine; and reference will be made to locals using positive offsets. The old value of the stack pointer will be restored on routine exit.

The main difficulty in the procedure call linkage is that the values passed must be in specific registers when routines are called or returned. A tentative assignment of locations:

Value	Location
sp	ib0.r63
ap	ib0.r62
ret-addr	ib0.r61
ret-val.I32	ib0.r60
ret-val.F32	fb0.r31
ret-val.F64	fb0.r31
ret-val.C64	fb0.r31,fb0.r30
ret-val.C128	fb0.r31,fb0.r30

(Is F32 the same as F64, and C64 the same as C128?) The correct sp must always be maintained in its assigned locations, so that interrupting routines may use it. The other values must be in their assigned location on a procedure entry or exit, as defined below. The location for a return value varies with the type of the result.

Each entry to a routine is marked with an ENTRY operation:

ENTRY entry-name ret-addr ap new-sp old-sp

The entry-name is a symbol identifying the entry, and is used for debugging purposes and the linker (and possible for other purposes). Ret-addr is the return address passed in from the caller. Ap is the argument pointer. Old-sp
is the caller's stack pointer. New-sp is this routine's stack pointer. Ret-addr, ⇐4.1
ap, and old-sp are in their assigned locations when the entry is executed. ⇐4.1

The amount of automatic storage needed by the routine will be available from the symbol table interface. Note that Phase 3 may need additional automatic storage, in which case the amount of automatic storage required may increase. Management of the spill area is left to the code generator. This could include allocating space in the spill area and managinga pointer into the spill area.

The ENTRY must expand into code which will save the old-sp, set the new-sp to $oldsp - reservation$ (where $reservation$ is the amount of automatic storage required by the routine), and store the new-sp into the assigned location. For routines with multiple entries, the names assigned ret-addr, ap, new-sp, old-sp should be the same in each entry. ⇐4.1

Routines are called with the LCALL (low-level CALL) operator:

lcall-oper	::=	LCALL.result-type result etemp new-ret-addr new-ap sp
result	::=	temp | ctemp | NIL
etemp	::=	rvalue

⇐4.1

Result-type is the type of the value returned by this entry to the routine; if no value is returned, VOID is used. Result-type must be one of VOID, I32, F32, F64, C64, or C128. The result temp is assigned the result returned by the routine. When result-type is VOID, result must be NIL. Ctemps are used for complex-valued routines (see definition of ctemp). ⇐4.1

Etemp is a 32 bit rvalue giving the address of the entry to be called. In most cases it will be a load time constant. ⇐4.1

(Note: The actual jump instruction used may turn out to be quite different when etemp is a temp as opposed to a constant. If this turns out to be inconvenient for the code generator, we may want to separate the two cases by providing two sets of operators, one for direct calls and one for indirect calls.) ⇐4.1

New-ret-addr is the address to return to, new-ap is the argument pointer for the called routine, and sp is the current stack pointer. The CALL operation must guarantee that new-ret-addr, new-ap, and sp are in their assigned location when control is transferred to the new routine. When the called function returns, the location assigned for return values of the appropriate type will hold the return value. ⇐4.1

(Note: We may wish to treat calls with no arguments specially. If so, an additional set of call operators may be added.)

Routines are exited with the RET operator: ⇐4.1

return-oper	::=	RET.result-type ret-value ret-addr old-sp
ret-value	::=	rvalue | crvalue | NIL

⇐4.1

Result-type is as defined above. Ret-value is the value returned by the routine; ret-addr is the return address; and old-sp is the stack pointer of the caller. The RET operation must guarantee ret-val and old-sp are in their assigned location, and then return control to the address in ret-addr.

An example:

```
ENTRY FOO ret-addr-foo ap-foo sp-foo sp-old
...
LCALL.I32  ret-val-bar bar-addr ret-addr-bar ap-bar sp-foo
...
RET.F64 ret-val-foo ret-addr-foo sp-old
```

(Code generator note: See TJK memo on procedure calls.)

2.7 Labels and jumps

Labels are not important in the IL-2. Label operands in branch instructions are only placeholders; the truth is in the flow graph. A LABEL pseudo-op for defining labels is used only for generating listings. The resolution of branch targets is a problem for Phase 3 and the assembler and loader. Similarly, direct GOTO's are not needed in IL-2 except for listings.

Labels which are the potential target of indirect jumps are an exeception to the above discussion. These labels are identified with a TAG pseudo-op:

TAG	label-name

These tagged labels are considered to be symbolic load time constants, and may be used as operands wherever a constant is permitted. In particular, the translation of the IL-1 LADDR operation will be an ASS.I32, assigning the tagged label to a temporary.

There is an IL-2 operation for performing an indirect jump:

IGOTO	temp

2.8 Conditionals

2.8.1 Conditional jumps

On the Saturn, a conditional jump is a two step process: a comparison or other operation sets a test bit, and a subsequent operation branches based on the setting of the test bit. The code generator will treat the test bits as resources, not as registers; they cannot be exposed in IL-2. Therefore the IL-2 conditional jump operations both perform a binary test and jump on the result.

The conditional operations all have the form:

$\boxed{\text{if-op temp rvalue label1 label2 prob}}$

Temp and rvalue are compared, using the operation indicated by the if-op. If the result is true, control is transferred to label1 else to label2. (Recall the labels are placeholders for flow graph edges.) Prob is the probability of branching to label1.

The conditional operations are: ⇐4.1

Integer	Floating	Logical	Bitwise
IF-EQ.32	unspecified	IF-GAND	unspecified
IF-NE.32		IF-GANDNL	
IF-LT.I32		IF-GANDNR	
IF-GE.I32		IF-GANDNB	
IF-LE.I32		IF-GOR	
IF-GT.I32		IF-GORNL	
IF-LT.SI32		IF-GORNR	
IF-GE.SI32		IF-GORNB	
IF-LE.SI32		IF-GEQV	
IF-GT.SI32		IF-GNEQV	

NL means 'not' the left operand; NR 'not' the right operand; NB 'not' both operands.

2.8.2 Select

For each conditional, there is a corresponding select operation. The select operations all have the form:

select-op rtemp stemp srvalue dtemp drvalue

Stemp and srvalue are compared, using the comparions indicated by the select-op. If the comparison is true, rtemp is assigned the result dtemp, else it is assigned drvalue.

The select operations are: ⇐4.1

Integer	Floating	Logical	Bitwise
SLCT-EQ.32	unspecified	SLCT-GAND	unspecified
SLCT-NE.32		SLCT-GANDNL	
SLCT-LT.I32		SLCT-GANDNR	
SLCT-GE.I32		SLCT-GANDNB	
SLCT-LE.I32		SLCT-GOR	
SLCT-GT.I32		SLCT-GORNL	
SLCT-LT.SI32		SLCT-GORNR	
SLCT-GE.SI32		SLCT-GORNB	
SLCT-LE.SI32		SLCT-GEQV	
SLCT-GT.SI32		SLCT-GNEQV	

2.9 Loop Control

Loops are entrances and edges are marked with pseudo-ops: ⇐4.1

LOOP-ENTRANCE	loop-id iteration-count
LOOP-EXIT	loop-id
BACK-EDGE	loop-id

Loop-id identifies the loop. Iteration count is the estimated iterations. (These operations may change as the needs of the trace picker and trace scheduler are known.)

2.10 Other Integer Operations

The remaining integer IL-2 operations map directly onto the machine instructions. They will change as the hardware changes. (In particular, the shifts and rotates are not yet pinned down.) Some integer instructions (e.g. multiply, divide) have not yet been defined, and are not included here. In all of the instructions, temp1 is assigned the result of a function applied to the remaining operands.

2.10.1 Extracts and merges

Operation		Description
EXT.UI8	temp1 temp2 rvalue	Extract (Unsigned 8 bit)
EXT.SI8	temp1 temp2 rvalue	Extract (Signed 8 bit)
EXT.UI16	temp1 temp2 rvalue	Extract (Unsigned 16 bit)
EXT.SI16	temp1 temp2 rvalue	Extract (Signed 16 bit)
MRG.UI8	temp1 temp2 rvalue1 rvalue2	Merge (Unsigned 8 bit)
MRG.SI8	temp1 temp2 rvalue1 rvalue2	Merge (Signed 8 bit)
MRG.UI16	temp1 temp2 rvalue1 rvalue2	Merge (Unsigned 16 bit)
MRG.SI16	temp1 temp2 rvalue1 rvalue2	Merge (Signed 16 bit)

2.10.2 Add and subtract

Operation		Description
ADD.U	temp1 temp2 rvalue	ADD (Unsigned)
ADD.S	temp1 temp2 rvalue	ADD (Signed)
ADDS2.U	temp1 temp2 rvalue	ADD Scaled by two (Unsigned)
ADDS4.U	temp1 temp2 rvalue	ADD Scaled by four (Unsigned)
ADDS8.U	temp1 temp2 rvalue	ADD Scaled by eight (Unsigned)
SUB.U	temp1 temp2 rvalue	SUBtract (Unsigned)
SUB.S	temp1 temp2 rvalue	SUBtract (Signed)

2.10.3 Multiply and divide

unspecified

2.10.4 Shifts and rotates

unspecified

2.10.5 "For value" comparisons

⇐4.1

Operation		Description
CEQ.I32	temp1 temp2 rvalue	Compare EQuals
CNE.I32	temp1 temp2 rvalue	Compare Not Equals
CLT.UI32	temp1 temp2 rvalue	Compare Less Than, unsigned
CGE.UI32	temp1 temp2 rvalue	Comp Greater than or Eq, unsigned
CLE.UI32	temp1 temp2 rvalue	Comp Less than or Eq, unsigned
CGT.UI32	temp1 temp2 rvalue	Compare Greater Than, unsigned
CLT.SI32	temp1 temp2 rvalue	Compare Less Than, signed
CGE.SI32	temp1 temp2 rvalue	Comp Greater than or Eq, signed
CLE.SI32	temp1 temp2 rvalue	Comp Less than or Eq, signed
CGT.SI32	temp1 temp2 rvalue	Compare Greater Than, signed

2.10.6 "For value" logicals

Operation		Description
GAND	temp1 temp2 rvalue	General AND
GOR	temp1 temp2 rvalue	General OR
GEQV	temp1 temp2 rvalue	General EQV

⇐4.1

(Note: This set may be expanded, depending on what the hardware provides.)

2.10.7 "For value" bitwise

unspecified

2.10.8 Miscellaneous

Operation		Description
GPSW	temp1 temp2 rvalue	Get Program Status Word
PPSW	temp1 temp2 rvalue	Put Program Status Word
BRV.UI32	temp1 temp2 rvalue	Bit ReVerse
FRO.UI32	temp1 temp2 rvalue	Find Rightmost One ??
FLO.UI32	temp1 temp2 rvalue	Find Leftmost One ??
AST.32	temp1 temp2 rvalue	Atomic Store, with word alignment
SAST.32	temp1 temp2 rvalue	Scaled Atomic Store, word align.

2.11 Other Floating Operations

$\boxed{\text{unspecified}}$