Iloc'93

Preston Briggs
Tim Harvey

CRPC-TR93323
July 1993

Center for Research on Parallel Computation
Rice University
P.O. Box 1892
Houston, TX 77251-1892
1 Introduction

ILOC '93 is the latest in a series of intermediate languages used for our Module Compiler. For convenience, we will simply use ILOC to indicate the current incarnation of the language.

ILOC shares the basic philosophy of earlier versions, but benefits from our experience. ILOC is a low-level language, allowing extensive optimization; in particular, we expect to be able to eliminate much of the overhead implicit in array indexing and procedure calls. On the other hand, ILOC is perhaps best suited for traditional optimizations, such as strength reduction, common subexpression elimination, value numbering, etc.; more aggressive dependence-based optimizations may require additional high-level structure.

The primary goal of this version of ILOC is to allow a greater degree of machine-independence in the optimizer than previously achieved. The primary changes are reflected in the handling of procedure calls. We believe that some portions of the front-end (a2i) are inherently machine dependent (especially in regard to calling conventions). Similarly, certain phases of the back-end are necessarily machine dependent. Nevertheless, we hope the bulk of the optimizer can be completely machine independent.

A secondary goal is to improve the readability of ILOC, both for humans and programs. Additionally, we wish to be able to record the results of analysis and optimization within the code. Changes from earlier versions include:

- The case of alphabetic characters is now significant.
- Opcodes have been renamed in a more systematic fashion.
- Register numbers are preceded with an 'r' to help emphasize the difference between registers and expressions.
- Provision is made for recording interprocedural information, in human-readable form, within the body of the procedure.
- Additional opcodes are defined to represent common addressing modes and loads from known constant memory.

An important non-goal is support for C. We think ILOC is adequate for good Fortran 77 compilers; however, it will need some extensions to support C. Weakness that come to mind include:

- inadequate support for character (byte) manipulation,
- no support for manipulation of short (16-bit) integers,
- no mechanism for representing pointer-based aliasing, and
- no support for volatile variables.

While none of these seem to insurmountable problems, we have not yet had occasion to extend ILOC in these directions.
2 An ILOC File

We expect that a single ILOC file will contain a single routine with associated data and declarations. The code (instructions) and the data are placed in separate segments. Executable code is placed in the program segment; data is placed in the static data segment. Data declarations may appear anywhere. Consider the following simple example:

```
foo:
   iDATA  0 100
   fDATA 3.14 1
bar:  1 FRAME 100 => r0 r1 r2 r3 r4
L123: 2 fADD r2 r3 => r5
      3 fRTW r0 r5 | 4 fSTOr @quux 4 0 r4 r5
dDATA 1.414 1
```

In this case, we see three constants placed in the static data segment and a trivial routine defined in the program segment. Segment control is implied the kind of statement. The data statements specify 100 integer zeroes, one single-precision floating-point constant equal to 3.14, and a double-precision floating-point constant equal to 1.414.

The executable operations each have statement numbers (hopefully referring to the original source text) to aid when reporting certain statistics. Notice that we can have more than one operation on a line. This implies that the two operations may be executed in parallel.

3 Labels

Instructions and declarations may have a symbolic label. A label is a string consisting of an alphabetic character followed by zero or more alphanumeric characters. Case is significant.

Some labels refer to routines or data areas defined in another compilation unit. Other labels may need to be visible to external routines. Such labels must be declared explicitly using the `NAME` declaration.

Often, labels will need to be invented to name new basic blocks created during the course of optimization. These will have the form:

```
Ldigits
```

4 Declarations

`NAME label` Declares `label` as visible to other routines. Basically used as a declaration to the linker.

`ALIAS var-tag [alias-tags]` The ALIAS instruction is a declaration which lists all of the aliases for a particular variable. Note that this is MAY-ALIAS information; that is, stores to one variable may affect an aliased variable. In the absence of any alias declarations, it is safe to assume that two variables are not aliased.

4.1 Variable Tags

The `tag` for a particular variable is simply the variable’s name, preceded by an “at sign” (the character @). For example,

```
quux    a variable name
@quux    the tag
```

Besides tag names derived from the program source, a2i may invent tags for its own use (for instance, to label locations in the stack frame). In the future, we may provide for more elaborate mechanisms to indicate array sections and so forth.
5 The Data Segment

The following instructions are used to initialize data within the static data segment. The value-expression specifies the value to be placed in memory; the repetition-expression specifies how many copies are to be placed in memory.

`bDATA integer-expression repetition-expression`

`iDATA integer-expression repetition-expression`

`fDATA single-expression repetition-expression`

`dDATA double-expression repetition-expression`

`BYTES size` Reserves size bytes of uninitialized storage. Typically used for allocating large arrays in COMMON blocks.

6 ILOC Instructions

We allow several operations to be packed into a single instruction, supporting experimentation with LIW architectures, branch-and-execute instructions, and auto-increment addressing modes. When an instruction is composed of several operations, the operations are (conceptually) issued in parallel; therefore, certain restrictions are implied:

- It doesn’t make sense to have more than 1 control-flow operation; i.e., no more than one jump, brach, return, or subroutine call per instruction.
- If several store operations are packed into an instruction, they should store to different locations.
- Similarly, no two operations should write to the same result register.

It is assumed that all inputs (memory locations and registers) are read before any results are written.

7 ILOC Operations

Registers in ILOC are not explicitly typed; however, if a floating-point operation attempts to use the result of an integer operation, it will make problems for the register allocator. Registers are numbered from zero and an infinite supply is assumed; however, a dense numbering is preferred (i.e., if using only 10 registers, number them 0 to 9).

In the spelling of opcodes (also labels and tags), case is significant; lower-case characters are used as modifiers for operations, which are defined to be in all upper-case. Generally, the “type” of each operation is indicated by a single lower-case letter preceding the opcode.

`b` byte or character
`i` integer
`f` single-precision floating-point `REAL*4`
`d` double-precision floating-point `REAL*8`
`c` single-precision complex `COMPLEX*8`
`q` double-precision complex `COMPLEX*16`

We have also defined address modifiers which are used to indicate addressing modes. When used, they appear at the end of the opcode.

`r` register
`l` label
`o` offset
Operations typically use and define registers. Used registers are always mentioned first. The string "=>" always appears before any defined registers. For example, in the statement

\[ \text{iADD } \text{r17 r18 } \text{=> r19} \]

the registers r17 and r18 are used to define r19.

The results of interprocedural analysis are represented via lists of tags. Items in the lists are separated by whitespace and the lists are enclosed in square brackets.

### 7.1 Miscellaneous

**HALT** The HALT operation is not executable. Instead, it serves as an indication to the optimizer that execution will never reach this point. Typically, it is emitted after a call to exit.

**NOP** A placeholder. Sometimes used for labels or to replace instructions that have been deleted.

### 7.2 Procedure Calls and Returns

**FRAME size => parms** The FRAME operation serves as a header for a subroutine. The size is an integer value specifying the size of the stack frame to be allocated. The parms list contains one or more registers whose value is known at the beginning of the subroutine. While the length and exact contents of this list are machine dependent, it will minimally contain the frame pointer and the registers containing parameters sent to the subroutine.

The JSRx operations specify a call (jump to subroutine) of the routine whose address is contained in the register reg. Registers containing arguments are specified by the parms list. All registers are preserved across the call.

The optional lists refs and mods contain the tags of variables possibly referenced or modified by the subroutine. For instance,

\[ \text{iJSRl foo r0 r10 r20 } \text{=> r30 [@bar] [@quux]} \]

indicates a call to the function foo, with the frame pointer in r0 and two parameters in r10 and r20. The result is returned in r30. The variable bar may be referenced and the variable quux may be modified.

If no interprocedural information is available, the lists will be absent. If no variables are referenced or modified, both lists will be empty. For example,

\[ \text{iJSRl foo r0 r10 r20 } \text{=> r30} \]
\[ \text{iJSRl bar r0 r10 r20 } \text{=> r30 [@quux] []} \]

No information is known about foo, so we must conservatively assume that all variables may be referenced and modified. On the other hand, we are certain that bar modifies no variables, though it may refer to quux.

**JSRx reg fp parms [refs] [mods]**

\[ \text{iJSR} \text{reg fp parms } \text{=> result [refs] [mods]} \] Call a function returning an INTEGER result in the register specified by result. This form is also used for calling CHAR and LOGICAL functions.

\[ \text{iJSR} \text{reg fp parms } \text{=> result [refs] [mods]} \] Call a function returning a REAL*4 result in the register specified by result.

\[ \text{iJSR} \text{reg fp parms } \text{=> result [refs] [mods]} \] Call a function returning a REAL*8 result in the register specified by result.

\[ \text{iJSR} \text{reg fp parms } \text{=> result [refs] [mods]} \] Call a function returning a COMPLEX*8 result in the register specified by result.

\[ \text{iJSR} \text{reg fp parms } \text{=> result [refs] [mods]} \] Call a function returning a COMPLEX*16 result in the register specified by result.
The JSR1 operations specify a call to a subroutine defined by \textit{label}. The list \textit{parms} is a list of registers containing the parameters to be sent to the subroutine. The lists \textit{refs} and \textit{mods} contain variables possibly referenced or modified by the subroutine. All registers are preserved across the call.

\textbf{JSR1} \hspace{.1cm} \textit{label \hspace{.1cm} fp \hspace{.1cm} parms \hspace{.1cm} [\textit{refs}] \hspace{.1cm} [\textit{mods}]} \hspace{.1cm} \textbf{JSR1} \hspace{.1cm} \textit{label \hspace{.1cm} fp \hspace{.1cm} parms \hspace{.1cm} => \hspace{.1cm} result \hspace{.1cm} [\textit{refs}] \hspace{.1cm} [\textit{mods}]} \hspace{.1cm} Call a function returning an \textbf{INTEGER} result in the register specified by \textit{result}. This form is also used for calling \textbf{CHAR} and \textbf{LOGICAL} functions. \hspace{.1cm} \textbf{JSR1} \hspace{.1cm} \textit{label \hspace{.1cm} fp \hspace{.1cm} parms \hspace{.1cm} => \hspace{.1cm} result \hspace{.1cm} [\textit{refs}] \hspace{.1cm} [\textit{mods}]} \hspace{.1cm} Call a function returning a \textbf{REAL*4} result in the register specified by \textit{result}. \hspace{.1cm} \textbf{JSR1} \hspace{.1cm} \textit{label \hspace{.1cm} fp \hspace{.1cm} parms \hspace{.1cm} => \hspace{.1cm} result \hspace{.1cm} [\textit{refs}] \hspace{.1cm} [\textit{mods}]} \hspace{.1cm} Call a function returning a \textbf{REAL*8} result in the register specified by \textit{result}. \hspace{.1cm} \textbf{JSR1} \hspace{.1cm} \textit{label \hspace{.1cm} fp \hspace{.1cm} parms \hspace{.1cm} => \hspace{.1cm} result \hspace{.1cm} [\textit{refs}] \hspace{.1cm} [\textit{mods}]} \hspace{.1cm} Call a function returning a \textbf{COMPLEX*8} result in the register specified by \textit{result}. \hspace{.1cm} \textbf{JSR1} \hspace{.1cm} \textit{label \hspace{.1cm} fp \hspace{.1cm} parms \hspace{.1cm} => \hspace{.1cm} result \hspace{.1cm} [\textit{refs}] \hspace{.1cm} [\textit{mods}]} \hspace{.1cm} Call a function returning a \textbf{COMPLEX*16} result in the register specified by \textit{result}.

The RTN operations specify a return from a subroutine call. \textit{fp} is the register containing the frame pointer. A list of live variables may be specified. If the list is not present, we must assume that all variables are live. If a list is present, we assume it is conservative and specifies at least all the live variables (typically members of common blocks and variables in the caller's stack frame).

\textbf{RTN} \hspace{.1cm} \textit{fp \hspace{.1cm} [\textit{live-vars}]} \hspace{.1cm} No value to be returned. \hspace{.1cm} \textbf{RTN} \hspace{.1cm} \textit{fp \hspace{.1cm} result \hspace{.1cm} [\textit{live-vars}]} \hspace{.1cm} INTEGER result to be returned in the \textit{result} register. \hspace{.1cm} \textbf{RTN} \hspace{.1cm} \textit{fp \hspace{.1cm} result \hspace{.1cm} [\textit{live-vars}]} \hspace{.1cm} REAL*4 result to be returned in the \textit{result} register. \hspace{.1cm} \textbf{RTN} \hspace{.1cm} \textit{fp \hspace{.1cm} result \hspace{.1cm} [\textit{live-vars}]} \hspace{.1cm} REAL*8 result to be returned in the \textit{result} register. \hspace{.1cm} \textbf{RTN} \hspace{.1cm} \textit{fp \hspace{.1cm} result \hspace{.1cm} [\textit{live-vars}]} \hspace{.1cm} COMPLEX*8 result to be returned in the \textit{result} register. \hspace{.1cm} \textbf{RTN} \hspace{.1cm} \textit{fp \hspace{.1cm} result \hspace{.1cm} [\textit{live-vars}]} \hspace{.1cm} COMPLEX*16 result to be returned in the \textit{result} register.

\textbf{7.3 Memory References}

Load and store operations refer to a single memory location at a time. Each operation is typed, indicating the size of the referenced location. Each operation contains a \textit{tag} (a constant string) whose value corresponds to the name of the source variable. Additionally, each operation contains an \textit{alignment} indicator (an integer constant) showing the minimal guaranteed alignment. Typical values would be 1, 2, 4, 8, 16, \ldots \, Normally, the compiler will align REAL*4 values to a four-byte boundary, REAL*8 values to an eight-byte boundary, and so forth. Occasionally, equivalence statements may force misalignment (or uncertain alignment).

We define a small hierarchy of load and store instructions. For the load instructions, the hierarchy looks like this:

\textbf{load immediate} \hspace{.1cm} Load a known value into a register. \hspace{.1cm} \textbf{load from constant memory} \hspace{.1cm} Load an unknown (but unchanging) value into a register. No subroutine or store can affect this value. \hspace{.1cm} \textbf{load from scalar memory} \hspace{.1cm} Load an unknown value from a scalar variable. No dependence analysis is required to build use-def chains for this variable. \hspace{.1cm} \textbf{load from array memory} \hspace{.1cm} Load an unknown value from a location in an array. It's generally impossible to build accurate use-def chains for array variables without dependence analysis.

For stores, the hierarchy is simpler since it only makes sense to store to scalar and array locations.
7.3.1 Load Immediate

iLDI \( \text{int.exp} \rightarrow \text{result} \) Load an immediate (INTEGER constant) value into a register.

fLDI \( \text{int.exp} \rightarrow \text{result} \) Convert an immediate (INTEGER constant) value to a REAL*4 and load it into the result register.

dLDI \( \text{int.exp} \rightarrow \text{result} \) Convert an immediate (INTEGER constant) value to a REAL*8 and load it into the result register.

cLDI \( \text{int.exp} \rightarrow \text{result} \) Convert an immediate (INTEGER constant) value to a COMPLEX*8 and load it into the result register. Note that the imaginary part of the complex value will be 0.

qLDI \( \text{int.exp} \rightarrow \text{result} \) Convert an immediate (INTEGER constant) value to a COMPLEX*16 and load it into the result register. Note that the imaginary part of the complex value will be 0.

7.3.2 Load from Constant Memory

The CON operations load a constant from memory. Only the "or" addressing form is expected to be required. While the tags are not expected to be useful for optimization, they will be used by the cache simulator.

bCONor \( \text{tag alignment offset base} \rightarrow \text{result} \) Load an unsigned byte constant from memory.

iCONor \( \text{tag alignment offset base} \rightarrow \text{result} \) Load an INTEGER constant from memory.

fCONor \( \text{tag alignment offset base} \rightarrow \text{result} \) Load a REAL*4 constant from memory.

dCONor \( \text{tag alignment offset base} \rightarrow \text{result} \) Load a REAL*8 constant from memory.

CCONor \( \text{tag alignment offset base} \rightarrow \text{result} \) Load a COMPLEX*8 constant from memory.

qCONor \( \text{tag alignment offset base} \rightarrow \text{result} \) Load a COMPLEX*16 constant from memory.

7.3.3 Load from Scalar Memory

A load operation copies a single value from memory into a result register.

The SLDo or operations load values from a memory location defined by the sum of a base register and an integer constant (the offset).

bSLDo or \( \text{tag alignment offset base} \rightarrow \text{result} \) Load an unsigned byte value from memory. The alignment is included since it might be possible to take advantage of the information to block together character operations.

iSLDo or \( \text{tag alignment offset base} \rightarrow \text{result} \) Load an INTEGER value from memory.

fSLDo or \( \text{tag alignment offset base} \rightarrow \text{result} \) Load a REAL*4 value from memory.

dSLDo or \( \text{tag alignment offset base} \rightarrow \text{result} \) Load a REAL*8 value from memory.

CSDLDo or \( \text{tag alignment offset base} \rightarrow \text{result} \) Load a COMPLEX*8 value from memory.

qSLDo or \( \text{tag alignment offset base} \rightarrow \text{result} \) Load a COMPLEX*16 value from memory.
The `SLDrr` operations load values from a memory location defined by the sum of a base register plus an index register.

- `bSLDrr` tag alignment index base => result Load an unsigned byte value from memory.
- `iSLDrr` tag alignment index base => result Load an INTEGER value from memory.
- `fSLDrr` tag alignment index base => result Load a REAL*4 value from memory.
- `dSLDrr` tag alignment index base => result Load a REAL*8 value from memory.
- `cSLDrr` tag alignment index base => result Load a COMPLEX*8 value from memory.
- `qSLDrr` tag alignment index base => result Load a COMPLEX*16 value from memory.

Note – Starred instructions are potentially machine dependent and should not be emitted by the front end. They are reserved for use by the optimizer and code generator.

### 7.3.4 Store to Scalar Memory

Store operations copy the contents of a value register into a memory location.

The `SSTor` operations store a value in the memory location defined by the sum of a base register and some integer constant offset.

- `bSSTor` tag alignment offset base value Store the byte value to memory.
- `iSSTor` tag alignment offset base value Store the INTEGER value to memory.
- `fSSTor` tag alignment offset base value Store the REAL*4 value to memory.
- `dSSTor` tag alignment offset base value Store the REAL*8 value to memory.
- `cSSTor` tag alignment offset base value Store the COMPLEX*8 value to memory.
- `qSSTor` tag alignment offset base value Store the COMPLEX*16 value to memory.

The `SSTrx` operations store a value in the memory location defined by the sum of the base register and the index register.

- `bSSTrx` tag alignment index base value Store the byte value to memory.
- `iSSTrx` tag alignment index base value Store the INTEGER value to memory.
- `fSSTrx` tag alignment index base value Store the REAL*4 value to memory.
- `dSSTrx` tag alignment index base value Store the REAL*8 value to memory.
- `cSSTrx` tag alignment index base value Store the COMPLEX*8 value to memory.
- `qSSTrx` tag alignment index base value Store the COMPLEX*16 value to memory.
7.3.5 Load from Array Memory

A load operation copies a single value from memory into a result register.

The LDr operations load values from a memory location defined by the sum of a base register and an integer constant (the offset).

bLDr tag alignment offset base => result Load an unsigned byte value from memory. The alignment is included since it might be possible to take advantage of the information in block address character operations.

iLDr tag alignment offset base => result Load an INTEGER value from memory.

fLDr tag alignment offset base => result Load a REAL*4 value from memory.

dLDr tag alignment offset base => result Load a REAL*8 value from memory.

cLDr tag alignment offset base => result Load a COMPLEX*8 value from memory.

qLDr tag alignment offset base => result Load a COMPLEX*16 value from memory.

The LDr operations load values from a memory location defined by the sum of a base register plus an index register.

bLDr tag alignment index base => result Load an unsigned byte value from memory.

iLDr tag alignment index base => result Load an INTEGER value from memory.

fLDr tag alignment index base => result Load a REAL*4 value from memory.

dLDr tag alignment index base => result Load a REAL*8 value from memory.

cLDr tag alignment index base => result Load a COMPLEX*8 value from memory.

qLDr tag alignment index base => result Load a COMPLEX*16 value from memory.

7.3.6 Store to Array Memory

Store operations copy the contents of a value register into a memory location.

The SDr operations store a value in the memory location defined by the sum of a base register and some integer constant offset.

bSDr tag alignment offset base value Store the byte value to memory.

iSDr tag alignment offset base value Store the INTEGER value to memory.

fSDr tag alignment offset base value Store the REAL*4 value to memory.

dSDr tag alignment offset base value Store the REAL*8 value to memory.

cSDr tag alignment offset base value Store the COMPLEX*8 value to memory.

qSDr tag alignment offset base value Store the COMPLEX*16 value to memory.

The SDr operations store a value in the memory location defined by the sum of the base register and the index register.

bSDr tag alignment index base value Store the byte value to memory.

iSDr tag alignment index base value Store the INTEGER value to memory.

fSDr tag alignment index base value Store the REAL*4 value to memory.

dSDr tag alignment index base value Store the REAL*8 value to memory.

cSDr tag alignment index base value Store the COMPLEX*8 value to memory.

qSDr tag alignment index base value Store the COMPLEX*16 value to memory.
7.4 Arithmetic

7.4.1 Integer Arithmetic

iADD addend addend => sum

iSUB minuend subtrahend => difference where difference ← minuend − subtrahend.

iMUL multiplicand multiplier => product

iDIV dividend divisor => quotient where quotient ← dividend/divisor.

iADDI immediate addend => sum

iSUBI immediate subtrahend => difference where difference ← immediate − subtrahend.

7.4.2 Single-Precision Arithmetic

fADD addend addend => sum

fSUB minuend subtrahend => difference

fMUL multiplicand multiplier => product

fDIV dividend divisor => quotient

7.4.3 Double-Precision Arithmetic

dADD addend addend => sum

dSUB minuend subtrahend => difference

dMUL multiplicand multiplier => product

dDIV dividend divisor => quotient

7.4.4 Single-Precision Complex Arithmetic

cADD addend addend => sum

cSUB minuend subtrahend => difference

cMUL multiplicand multiplier => product

cDIV dividend divisor => quotient

7.4.5 Double-Precision Complex Arithmetic

qADD addend addend => sum

qSUB minuend subtrahend => difference

qMUL multiplicand multiplier => product

qDIV dividend divisor => quotient
7.5 Control Flow

**JMP** label Jump to the location of label.

**JMP** reg [labels] Jump to the location pointed to by reg. The possible values of reg are indicated by the list labels. For best results, the label list should be as precise as possible; extra labels will cause extra edges in the control-flow graph and severely degrade optimization.

**BR true false op** Branch to the true label if the boolean value in op is TRUE, else branch to the false label.

7.6 Comparisons and Conditional Branches

The handling of compares and branches is difficult to accomplish in a machine-independent fashion. There seem to be two styles used in common machines:

1. A general compare (or perhaps subtract) setting a condition code, followed by a specific branch on a particular condition code setting (e.g., branch if greater-than).

2. A specific compare (e.g., testing for equality) setting a result register to true or false, then branching based on the sense of the result register.

We try to support both schemes. Typically, a comparison and branch will require three operations. The first is a compare which sets a result to -1, 0, or 1 if the first operand is less than, equal to, or greater than the second operand, respectively. The second operation is a logical operation which reads the result of the comparison and returns a boolean result for the specified condition. The final operation would be a branch on the sense of the logical operation. During optimization, we may combine two of these operations, depending on which scheme the target machine supports.

**CMP** operations compare the values in two registers and put the result into the result register cc. Note that the result register is not a special machine register; rather, it is simply a general-purpose register.

\[
iCMP \text{ op1 op2 } \rightarrow cc \text{ Compare two INTEGER values.}
\]

\[
fCMP \text{ op1 op2 } \rightarrow cc \text{ Compare two REAL*4 values.}
\]

\[
dCMP \text{ op1 op2 } \rightarrow cc \text{ Compare the REAL*8 values.}
\]

\[
cCMP \text{ op1 op2 } \rightarrow cc \text{ Compare the COMPLEX*8 values.}
\]

\[
qCMP \text{ op1 op2 } \rightarrow cc \text{ Compare the COMPLEX*16 values.}
\]

The following operations take the result of one of the CMP operations (above) and convert it into a boolean value. The possible values are TRUE and FALSE, although the implementation of these values will be with the integers -1 and 0, respectively.

\[
EQ \text{ cc } \rightarrow \text{ logical Set logical to TRUE if cc } = 0, \text{ FALSE otherwise.}
\]

\[
NE \text{ cc } \rightarrow \text{ logical Set logical to TRUE if cc } \neq 0, \text{ FALSE otherwise.}
\]

\[
LE \text{ cc } \rightarrow \text{ logical Set logical to TRUE if cc } \neq 1, \text{ FALSE otherwise.}
\]

\[
GE \text{ cc } \rightarrow \text{ logical Set logical to TRUE if cc } \neq -1, \text{ FALSE otherwise.}
\]

\[
LT \text{ cc } \rightarrow \text{ logical Set logical to TRUE if cc } = -1, \text{ FALSE otherwise.}
\]

\[
GT \text{ cc } \rightarrow \text{ logical Set logical to TRUE if cc } = 1, \text{ FALSE otherwise.}
\]
The CMPcc instructions compare two operands, testing for a specific condition, and set the result register logical to either TRUE or FALSE. Effectively, the iCMPeq instruction serves as a combination of the iCMP instruction and the EQ instruction.

\[
\text{iCMPeq op1 op2 => logical} \\
\text{iCMPne op1 op2 => logical} \\
\text{iCMPle op1 op2 => logical} \\
\text{iCMPge op1 op2 => logical} \\
\text{iCMPlt op1 op2 => logical} \\
\text{iCMPlt op1 op2 => logical}
\]

\[
\text{fCMPeq op1 op2 => logical} \\
\text{fCMPne op1 op2 => logical} \\
\text{fCMPle op1 op2 => logical} \\
\text{fCMPge op1 op2 => logical} \\
\text{fCMPlt op1 op2 => logical} \\
\text{fCMPlt op1 op2 => logical}
\]

\[
\text{dCMPeq op1 op2 => logical} \\
\text{dCMPne op1 op2 => logical} \\
\text{dCMPle op1 op2 => logical} \\
\text{dCMPge op1 op2 => logical} \\
\text{dCMPlt op1 op2 => logical} \\
\text{dCMPlt op1 op2 => logical}
\]

\[
\text{cCMPeq op1 op2 => logical} \\
\text{cCMPne op1 op2 => logical}
\]

\[
\text{qCMPeq op1 op2 => logical} \\
\text{qCMPne op1 op2 => logical}
\]

The BRcc operations branch to the true label if the appropriate test on the cc register is TRUE, otherwise they branch to the false label. Effectively, a BReq combines an EQ instruction and a BR instruction.

\[
\text{BReq true false cc} \\
\text{BRne true false cc} \\
\text{BRle true false cc} \\
\text{BRge true false cc} \\
\text{BRlt true false cc} \\
\text{BRgt true false cc}
\]
7.7 Intrinsics

7.7.1 Conversions and Copies

The conversion functions are not specified completely by the Fortran ANSI standard, so we have defined our own. All of the function names are built with the following formula:

<old_type>*2<new_type>

where old_type and new_type are taken from the set of type prefixes for non-intrinsic iloc operations.

i2i old => new Copy.
i2f old => new
i2d old => new
i2c old => new Assigns (old, 0) to new.
i2q old => new Assigns (old, 0) to new.

f2i old => new Truncation.
f2f old => new Copy.
f2d old => new
f2c old => new Assigns (old, 0) to new.
f2q old => new Assigns (old, 0) to new.

d2i old => new Truncation.
d2f old => new
d2d old => new Copy.
d2c old => new Assigns (old, 0) to new.
d2q old => new Assigns (old, 0) to new.

c2i old => new Returns the truncated real part.
c2f old => new Returns the real part.
c2d old => new Returns the real part.
c2c old => new Copy.
c2q old => new

q2i old => new Returns the truncated real part.
q2f old => new Returns the real part.
q2d old => new Returns the real part.
q2c old => new
q2q old => new Copy.
cCOMPLEX real.1 real.2 => complex_result Construct a COMPLEX•8
qCOMPLEX double.1 double.2 => dcomplex_result Construct a COMPLEX•16

7.7.2 Intrinsic Functions

Truncation

fTRUNC real.input => real.result Implements AINT()
dTRUNC double.input => double.result Implements DINT()

Rounding

fROUND real.input => real.result Implements ANINT()
dROUND double.input => double.result Implements DNINT()
fNINT real.input => integer_result Implements NINT()
dNINT double.input => integer_result Implements DNINT()

Absolute Value

iABS integer.input => integer.result Implements IABS()
fABS real.input => real.result Implements ABS()
dABS double.input => double.result Implements DABS()
cABS complex.input => real.result The absolute value of a complex is $\sqrt{\text{real.part}^2 + \text{imag.part}^2}$ and yields a real value, not a complex one.
qABS complex.input => double.result The absolute value of a complex is $\sqrt{\text{real.part}^2 + \text{imag.part}^2}$ and yields a real value, not a complex one.

Arithmetic Shifts Typically, integer shift operations are emitted by the optimizer to obtain quicker computation of multiply and divide. Note that while shifting a positive integer right one place is equivalent to dividing by two, the same trick does not work for negative numbers.

iSL amount.reg value.reg => result Shift left. The shift amount in amount.reg must be $\geq 0$. *
iSLI amount.exp value.reg => result Shift left. The shift amount specified by amount.exp must be $\geq 0$. *
iSR amount.reg value.reg => result Shift right. The shift amount in amount.reg must be $\geq 0$. *
iSRI amount.exp value.reg => result Shift right. The shift amount specified by amount.exp must be $\geq 0$. *
Bitwise Logical Shifts  Bitwise shifts (also commonly known as logical shifts) are used for bit-field manipulations. The primary difference from integer shifts is that bitwise right shifts move 0 bits into the the most significant bit position; integer shifts duplicate the value of the sign bit, thus preserving the sign of integer values.

1SHIFT amount_reg value_reg => result Shift the value contained in value_reg by the amount contained in the amount_reg. The shift amount may be positive, negative, or zero. A negative value indicates a right shift and a positive value indicates a left shift. This instruction is generated by the front end and will be subjected to optimization. If necessary, it will be converted to one of the following instructions before code generation.

1SLI amount_reg value_reg => result Shift left. The shift amount in amount_reg must be ≥ 0.  *
1SLI amount_exp value_reg => result Shift left. The shift amount specified by amount_exp must be ≥ 0.  *
1SR amount_reg value_reg => result Shift Right. The shift amount in amount_reg must be ≥ 0.  *
1SRI amount_exp value_reg => result Shift right. The shift amount specified by amount_exp must be ≥ 0.  *

Bitwise Logical Operations  The following operations perform the appropriate logical operation in a bitwise fashion between two integer values, putting the result into the result register.

1AND op1 op2 => result
1OR op1 op2 => result
1XOR op1 op2 => result
1NOT op => result

Remainder

1MOD integer_dividend integer_divisor => integer_remainder Implements MOD()
1REM real_dividend real_divisor => real_remainder Implements AMOD()
1REM double_dividend double_divisor => double_remainder Implements DMOD()

Sign Transfer

1SIGN integer_input sign_determining_integer => integer_result Implements ISIGN()  
1SIGN real_input sign_determining_real => real_result Implements SIGN()  
1SIGN double_input sign_determining_double => double_result Implements DSIGN()  

Positive Difference  If minuend > subtrahend, the result is minuend − subtrahend. Otherwise, the result is zero.

1DIM integer_minuend integer_subtrahend => integer_positive_difference
1DIM real_minuend real_subtrahend => real_positive_difference
1DIM double_minuend double_subtrahend => double_positive_difference
Double Precision Product

\[ \text{dPROD} \text{ real\_multiplicand real\_multiplier} \Rightarrow \text{double\_product} \]

Maximum

\[ \text{iMAX} \text{ integer\_1 integer\_2} \Rightarrow \text{integer\_result} \]
\[ \text{fMAX} \text{ real\_1 real\_2} \Rightarrow \text{real\_result} \]
\[ \text{dMAX} \text{ double\_1 double\_2} \Rightarrow \text{double\_result} \]

Minimum

\[ \text{iMIN} \text{ integer\_1 integer\_2} \Rightarrow \text{integer\_result} \]
\[ \text{fMIN} \text{ real\_1 real\_2} \Rightarrow \text{real\_result} \]
\[ \text{dMIN} \text{ double\_1 double\_2} \Rightarrow \text{double\_result} \]

Imaginary Part of Complex

\[ \text{cIMAG complex\_input} \Rightarrow \text{real\_result} \text{ Extracts the imaginary part of a COMPLEX*8} \]
\[ \text{qIMAG dcomplex\_input} \Rightarrow \text{double\_result} \text{ Extracts the imaginary part of a COMPLEX*16} \]

Conjugate \( (r, i) \Rightarrow (r, -i) \)

\[ \text{cCONJ complex\_input} \Rightarrow \text{complex\_result} \text{ For COMPLEX*8} \]
\[ \text{qCONJ dcomplex\_input} \Rightarrow \text{dcomplex\_result} \text{ For COMPLEX*16} \]

Square Root

\[ \text{fSQR} \text{ real\_input} \Rightarrow \text{real\_result} \]
\[ \text{dSQR} \text{ double\_input} \Rightarrow \text{double\_result} \]
\[ \text{cSQR} \text{ complex\_input} \Rightarrow \text{complex\_result} \]
\[ \text{qSQR} \text{ dcomplex\_input} \Rightarrow \text{dcomplex\_result} \]

Exponential \( \text{Complements } e^x \)

\[ \text{fEXP} \text{ real\_exponent} \Rightarrow \text{real\_result} \]
\[ \text{dEXP} \text{ double\_exponent} \Rightarrow \text{double\_result} \]
\[ \text{cEXP} \text{ complex\_exponent} \Rightarrow \text{complex\_result} \]
\[ \text{qEXP} \text{ dcomplex\_exponent} \Rightarrow \text{dcomplex\_result} \]

Natural Logarithm

\[ \text{fLOG} \text{ real\_antilog} \Rightarrow \text{real\_logarithm} \]
\[ \text{dLOG} \text{ double\_antilog} \Rightarrow \text{double\_logarithm} \]
\[ \text{cLOG} \text{ complex\_antilog} \Rightarrow \text{complex\_logarithm} \]
\[ \text{qLOG} \text{ dcomplex\_antilog} \Rightarrow \text{dcomplex\_logarithm} \]
Common Logarithm  Computes the logarithm, base 10

fLOG10  real_antilog => real_logarithm

dLOG10  double_antilog => double_logarithm

Exponentiation  These are provided to handle FORTRAN’s exponentiation operator. They each compute \( \text{input}_1 \times \text{input}_2 \).

iPOW  integer_input1 integer_input2 => integer_result

fPOW  float_input1 float_input2 => float_result

dPOW  double_input1 double_input2 => double_result

cPOW  complex_input1 complex_input2 => complex_result

qPOW  dcomplex_input1 dcomplex_input2 => dcomplex_result

Since raising a number to an exact integer power is an important special case, we provide separate operations to handle it. Of course, the case of an integer raised to an integer power is already handled above.

fPOWi  float_input integer_input => float_result

dPOWi  double_input integer_input => double_result

cPOWi  complex_input integer_input => complex_result

qPOWi  dcomplex_input integer_input => dcomplex_result

Sine

fSIN  real_input => real_result

dSIN  double_input => double_result

cSIN  complex_input => complex_result

qSIN  dcomplex_input => dcomplex_result

Cosine

fCOS  real_input => real_result

dCOS  double_input => double_result

cCOS  complex_input => complex_result

qCOS  dcomplex_input => dcomplex_result

Tangent

fTAN  real_input => real_result

dTAN  double_input => double_result

Arcsine

fASIN  real_input => real_result

dASIN  double_input => double_result
Arcosine
\[ \text{fACOS} \text{ real\_input} => \text{real\_result} \]
\[ \text{dACOS} \text{ double\_input} => \text{double\_result} \]

Arctangent
\[ \text{fATAN} \text{ real\_input} => \text{real\_result} \]
\[ \text{dATAN} \text{ double\_input} => \text{double\_result} \]
Note that \text{xATAN2} produces the arctan of a quotient, where the divisor may be zero.
\[ \text{fATAN2} \text{ real\_dividend real\_divisor} => \text{real\_result} \]
\[ \text{dATAN2} \text{ double\_dividend double\_divisor} => \text{double\_result} \]

Hyperbolic Sine
\[ \text{fSINH} \text{ real\_input} => \text{real\_result} \]
\[ \text{dSINH} \text{ double\_input} => \text{double\_result} \]

Hyperbolic Cosine
\[ \text{fCOSH} \text{ real\_input} => \text{real\_result} \]
\[ \text{dCOSH} \text{ double\_input} => \text{double\_result} \]

Hyperbolic Tangent
\[ \text{fTANH} \text{ real\_input} => \text{real\_result} \]
\[ \text{dTANH} \text{ double\_input} => \text{double\_result} \]

8 Assertions
Assertions may appear in the code to indicate that certain facts hold at that particular point. They are not executable; rather, they are to act as hints to the optimizer (typically, they are inserted by one pass of the optimizer as a hint to another pass of the optimizer). On the other hand, they cannot simply be ignored since they often imply a copy (or renaming) from a source to the result.
\[ \text{iASRT} \text{ immediate source} => \text{result} \text{ Asserts that the result register is equal to the integer expression.} \]
\[ \text{iASRTeq} \text{ source1 source2} => \text{result} \text{ This operation asserts that the integer source registers have the same value.} \]
\[ \text{fASRTeq} \text{ source1 source2} => \text{result} \text{ This operation asserts that the floating-point source registers have the same value.} \]
\[ \text{dASRTeq} \text{ source1 source2} => \text{result} \text{ This operation asserts that the double-precision source registers have the same value.} \]
\[ \text{cASRTeq} \text{ source1 source2} => \text{result} \text{ This operation asserts that the complex source registers have the same value.} \]
\[ \text{qASRTeq} \text{ source1 source2} => \text{result} \text{ This operation asserts that the double-complex source registers have the same value.} \]