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

Iloc '93

Preston Briggs
preston@cs.rice.edu

Tim Harvey
harv@cs.rice.due

July 6, 1993

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     fRTN   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:

L*digits*

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 explicitely 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

```
iADD r17 r18 => 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 `JSRr` 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,

```
iJSRl foo r0 r10 r20 => 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,

```
iJSRl foo r0 r10 r20 => r30
iJSRl bar r0 r10 r20 => 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`.

`JSRr` *reg fp parms* `[`*refs*`]` `[`*mods*`]`

`iJSRr` *reg fp parms* `=>` *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.

`fJSRr` *reg fp parms* `=>` *result* `[`*refs*`]` `[`*mods*`]` Call a function returning a `REAL*4` result in the register specified by *result*.

`dJSRr` *reg fp parms* `=>` *result* `[`*refs*`]` `[`*mods*`]` Call a function returning a `REAL*8` result in the register specified by *result*.

`cJSRr` *reg fp parms* `=>` *result* `[`*refs*`]` `[`*mods*`]` Call a function returning a `COMPLEX*8` result in the register specified by *result*.

`qJSRr` *reg fp parms* `=>` *result* `[`*refs*`]` `[`*mods*`]` Call a function returning a `COMPLEX*16` result in the register specified by *result*.

The `JSRl` operations specify a call to a subroutine defined by *label*. The list *parms* is a list of registers containing the parameters to be sent to the subroutine. The lists *refs* and *mods* contain variables possibly referenced or modified by the subroutine. All registers are preserved across the call.

`JSRl` *label fp parms* `[`*refs*`]` `[`*mods*`]`

`iJSRl` *label fp parms* `=>` *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.

`fJSRl` *label fp parms* `=>` *result* `[`*refs*`]` `[`*mods*`]` Call a function returning a `REAL*4` result in the register specified by *result*.

`dJSRl` *label fp parms* `=>` *result* `[`*refs*`]` `[`*mods*`]` Call a function returning a `REAL*8` result in the register specified by *result*.

`cJSRl` *label fp parms* `=>` *result* `[`*refs*`]` `[`*mods*`]` Call a function returning a `COMPLEX*8` result in the register specified by *result*.

`qJSRl` *label fp parms* `=>` *result* `[`*refs*`]` `[`*mods*`]` Call a function returning a `COMPLEX*16` result in the register specified by *result*.

The `RTN` operations specify a return from a subroutine call. *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).

`RTN` *fp* `[`*live-vars*`]` No value to be returned.

`iRTN` *fp result* `[`*live-vars*`]` `INTEGER` result to be returned in the *result* register.

`fRTN` *fp result* `[`*live-vars*`]` `REAL*4` result to be returned in the *result* register.

`dRTN` *fp result* `[`*live-vars*`]` `REAL*8` result to be returned in the *result* register.

`cRTN` *fp result* `[`*live-vars*`]` `COMPLEX*8` result to be returned in the *result* register.

`qRTN` *fp result* `[`*live-vars*`]` `COMPLEX*16` result to be returned in the *result* register.

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 *tag* (a constant string) whose value corresponds to the name of the source variable. Additionally, each operation contains an *alignment* indicator (an integer constant) showing the minimal guaranteed alignment. Typical values would be 1, 2, 4, 8, 16, ... 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:

load immediate Load a known value into a register.

load from constant memory Load an unknown (but unchanging) value into a register. No subroutine or store can affect this value.

load from scalar memory Load an unknown value from a scalar variable. No dependence analysis is required to build use-def chains for this variable.

load from array memory 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` *int_exp* `=>` *result* Load an immediate (`INTEGER` constant) value into a register.

`fLDI` *int_exp* `=>` *result* Convert an immediate (`INTEGER` constant) value to a `REAL*4` and load it into the *result* register.

`dLDI` *int_exp* `=>` *result* Convert an immediate (`INTEGER` constant) value to a `REAL*8` and load it into the *result* register.

`cLDI` *int_exp* `=>` *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` *int_exp* `=>` *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` *tag alignment offset base* `=>` *result* Load an unsigned byte constant from memory.

`iCONor` *tag alignment offset base* `=>` *result* Load an `INTEGER` constant from memory.

`fCONor` *tag alignment offset base* `=>` *result* Load a `REAL*4` constant from memory.

`dCONor` *tag alignment offset base* `=>` *result* Load a `REAL*8` constant from memory.

`cCONor` *tag alignment offset base* `=>` *result* Load a `COMPLEX*8` constant from memory.

`qCONor` *tag alignment offset base* `=>` *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 `SLDor` operations load values from a memory location defined by the sum of a *base* register and an integer constant (the *offset*).

`bSLDor` *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 to block together character operations.

`iSLDor` *tag alignment offset base* `=>` *result* Load an `INTEGER` value from memory.

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

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

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

`qSLDor` *tag alignment offset base* `=>` *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 `SSTrr` operations store a value in the memory location defined by the sum of the *base* register and the *index* register.

`bSSTrr` *tag alignment index base value* Store the byte value to memory. ⋆

`iSSTrr` *tag alignment index base value* Store the `INTEGER` value to memory. ⋆

`fSSTrr` *tag alignment index base value* Store the `REAL*4` value to memory. ⋆

`dSSTrr` *tag alignment index base value* Store the `REAL*8` value to memory. ⋆

`cSSTrr` *tag alignment index base value* Store the `COMPLEX*8` value to memory. ⋆

`qSSTrr` *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 `LDor` operations load values from a memory location defined by the sum of a *base* register and an integer constant (the *offset*).

`bLDor` *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 to block together character operations.

`iLDor` *tag alignment offset base* `=>` *result* Load an `INTEGER` value from memory.

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

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

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

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

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

`bLDrr` *tag alignment index base* `=>` *result* Load an unsigned byte value from memory. ⋆

`iLDrr` *tag alignment index base* `=>` *result* Load an `INTEGER` value from memory. ⋆

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

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

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

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

`bSTor` *tag alignment offset base value* Store the byte value to memory.

`iSTor` *tag alignment offset base value* Store the `INTEGER` value to memory.

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

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

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

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

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

`bSTrr` *tag alignment index base value* Store the byte value to memory. ⋆

`iSTrr` *tag alignment index base value* Store the `INTEGER` value to memory. ⋆

`fSTrr` *tag alignment index base value* Store the `REAL*4` value to memory. ⋆

`dSTrr` *tag alignment index base value* Store the `REAL*8` value to memory. ⋆

`cSTrr` *tag alignment index base value* Store the `COMPLEX*8` value to memory. ⋆

`qSTrr` *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

`JMPl` *label* Jump to the location of *label*.

`JMPr` *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 severly 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` *op1 op2* `=>` *cc* Compare two `INTEGER` values.

`fCMP` *op1 op2* `=>` *cc* Compare two `REAL*4` values.

`dCMP` *op1 op2* `=>` *cc* Compare the `REAL*8` values.

`cCMP` *op1 op2* `=>` *cc* Compare the `COMPLEX*8` values.

`qCMP` *op1 op2* `=>` *cc* 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` *cc* `=>` *logical* Set *logical* to `TRUE` if $cc = 0$, `FALSE` otherwise.

`NE` *cc* `=>` *logical* Set *logical* to `TRUE` if $cc \neq 0$, `FALSE` otherwise.

`LE` *cc* `=>` *logical* Set *logical* to `TRUE` if $cc \neq 1$, `FALSE` otherwise.

`GE` *cc* `=>` *logical* Set *logical* to `TRUE` if $cc \neq -1$, `FALSE` otherwise.

`LT` *cc* `=>` *logical* Set *logical* to `TRUE` if $cc = -1$, `FALSE` otherwise.

`GT` *cc* `=>` *logical* Set *logical* to `TRUE` if $cc = 1$, `FALSE` otherwise.

The `CMP`*cc* 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.

`iCMPeq` *op1 op2* `=>` *logical* ⋆

`iCMPne` *op1 op2* `=>` *logical* ⋆

`iCMPle` *op1 op2* `=>` *logical* ⋆

`iCMPge` *op1 op2* `=>` *logical* ⋆

`iCMPlt` *op1 op2* `=>` *logical* ⋆

`iCMPgt` *op1 op2* `=>` *logical* ⋆

`fCMPeq` *op1 op2* `=>` *logical* ⋆

`fCMPne` *op1 op2* `=>` *logical* ⋆

`fCMPle` *op1 op2* `=>` *logical* ⋆

`fCMPge` *op1 op2* `=>` *logical* ⋆

`fCMPlt` *op1 op2* `=>` *logical* ⋆

`fCMPgt` *op1 op2* `=>` *logical* ⋆

`dCMPeq` *op1 op2* `=>` *logical* ⋆

`dCMPne` *op1 op2* `=>` *logical* ⋆

`dCMPle` *op1 op2* `=>` *logical* ⋆

`dCMPge` *op1 op2* `=>` *logical* ⋆

`dCMPlt` *op1 op2* `=>` *logical* ⋆

`dCMPgt` *op1 op2* `=>` *logical* ⋆

`cCMPeq` *op1 op2* `=>` *logical* ⋆

`cCMPne` *op1 op2* `=>` *logical* ⋆

`qCMPeq` *op1 op2* `=>` *logical* ⋆

`qCMPne` *op1 op2* `=>` *logical* ⋆

The `BR`*cc* 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.

`BReq` *true false cc* ⋆

`BRne` *true false cc* ⋆

`BRle` *true false cc* ⋆

`BRge` *true false cc* ⋆

`BRlt` *true false cc* ⋆

`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 `IDNINT()`

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 $\sqrt{real\_part^2 + imag\_part^2}$ and yields a real value, not a complex one.

`qABS` *dcomplex_input* `=>` *double_result* The absolute value of a complex is $\sqrt{real\_part^2 + imag\_part^2}$ 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 ≥ 0. ⋆

`iSLI` *amount_exp value_reg* `=>` *result* Shift left. The shift amount specified by *amount_exp* must be ≥ 0. ⋆

`iSR` *amount_reg value_reg* `=>` *result* Shift right. The shift amount in *amount_reg* must be ≥ 0. ⋆

`iSRI` *amount_exp value_reg* `=>` *result* Shift right. The shift amount specified by *amount_exp* must be ≥ 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.

`lSHIFT` *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.

`lSL` *amount_reg value_reg* `=>` *result* Shift left. The shift amount in *amount_reg* must be ≥ 0. $\star$

`lSLI` *amount_exp value_reg* `=>` *result* Shift left. The shift amount specified by *amount_exp* must be ≥ 0. $\star$

`lSR` *amount_reg value_reg* `=>` *result* Shift Right. The shift amount in *amount_reg* must be ≥ 0. $\star$

`lSRI` *amount_exp value_reg* `=>` *result* Shift right. The shift amount specified by *amount_exp* must be ≥ 0. $\star$

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.

`lAND` *op1 op2* `=>` *result*

`lOR` *op1 op2* `=>` *result*

`lNAND` *op1 op2* `=>` *result*

`lNOR` *op1 op2* `=>` *result*

`lEQV` *op1 op2* `=>` *result*

`lXOR` *op1 op2* `=>` *result*

`lNOT` *op* `=>` *result*

Remainder

`iMOD` *integer_dividend integer_divisor* `=>` *integer_remainder* Implements `MOD()`

`fMOD` *real_dividend real_divisor* `=>` *real_remainder* Implements `AMOD()`

`dMOD` *double_dividend double_divisor* `=>` *double_remainder* Implements `DMOD()`

Sign Transfer

`iSIGN` *integer_input sign_determining_integer* `=>` *integer_result* Implements `ISIGN()`

`fSIGN` *real_input sign_determining_real* `=>` *real_result* Implements `SIGN()`

`dSIGN` *double_input sign_determining_double* `=>` *double_result* Implements `DSIGN()`

Positive Difference If *minuend* $>$ *subtrahend*, the result is *minuend* $-$ *subtrahend*. Otherwise, the result is zero.

`iDIM` *integer_minuend integer_subtrahend* `=>` *integer_positive_difference*

`fDIM` *real_minuend real_subtrahend* `=>` *real_positive_difference*

`dDIM` *double_minuend double_subtrahend* `=>` *double_positive_difference*

Double Precision Product

`dPROD` *real_multiplicand real_multiplier* `=>` *double_product*

Maximum

`iMAX` *integer_1 integer_2* `=>` *integer_result*

`fMAX` *real_1 real_2* `=>` *real_result*

`dMAX` *double_1 double_2* `=>` *double_result*

Minimum

`iMIN` *integer_1 integer_2* `=>` *integer_result*

`fMIN` *real_1 real_2* `=>` *real_result*

`dMIN` *double_1 double_2* `=>` *double_result*

Imaginary Part of Complex

`cIMAG` *complex_input* `=>` *real_result* Extracts the imaginary part of a `COMPLEX*8`

`qIMAG` *dcomplex_input* `=>` *double_result* Extracts the imaginary part of a `COMPLEX*16`

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

`cCONJ` *complex_input* `=>` *complex_result* For `COMPLEX*8`

`qCONJ` *dcomplex_input* `=>` *dcomplex_result* For `COMPLEX*16`

Square Root

`fSQRT` *real_input* `=>` *real_result*

`dSQRT` *double_input* `=>` *double_result*

`cSQRT` *complex_input* `=>` *complex_result*

`qSQRT` *dcomplex_input* `=>` *dcomplex_result*

Exponential Computes e^x

`fEXP` *real_exponent* `=>` *real_result*

`dEXP` *double_exponent* `=>` *double_result*

`cEXP` *complex_exponent* `=>` *complex_result*

`qEXP` *dcomplex_exponent* `=>` *dcomplex_result*

Natural Logarithm

`fLOG` *real_antilog* `=>` *real_logarithm*

`dLOG` *double_antilog* `=>` *double_logarithm*

`cLOG` *complex_antilog* `=>` *complex_logarithm*

`qLOG` *dcomplex_antilog* `=>` *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 $input_1 ** input_2$.

`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*

Arccosine

`fACOS` *real_input* `=>` *real_result*

`dACOS` *double_input* `=>` *double_result*

Arctangent

`fATAN` *real_input* `=>` *real_result*

`dATAN` *double_input* `=>` *double_result*

Note that `xATAN2` produces the arctan of a quotient, where the divisor may be zero.

`fATAN2` *real_dividend real_divisor* `=>` *real_result*

`dATAN2` *double_dividend double_divisor* `=>` *double_result*

Hyperbolic Sine

`fSINH` *real_input* `=>` *real_result*

`dSINH` *double_input* `=>` *double_result*

Hyperbolic Cosine

`fCOSH` *real_input* `=>` *real_result*

`dCOSH` *double_input* `=>` *double_result*

Hyperbolic Tangent

`fTANH` *real_input* `=>` *real_result*

`dTANH` *double_input* `=>` *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.

`iASRT` *immediate source* `=>` *result* Asserts that the *result* register is equal to the integer expression. ⋆

`iASRTeq` *source1 source2* `=>` *result* This operation asserts that the integer source registers have the same value. ⋆

`fASRTeq` *source1 source2* `=>` *result* This operation asserts that the floating-point source registers have the same value. ⋆

`dASRTeq` *source1 source2* `=>` *result* This operation asserts that the double-precision source registers have the same value. ⋆

`cASRTeq` *source1 source2* `=>` *result* This operation asserts that the complex source registers have the same value. ⋆

`qASRTeq` *source1 source2* `=>` *result* This operation asserts that the double-complex source registers have the same value. ⋆