The Massively Scalar Compiler Project

Preston Briggs

September 6, 1996

[0]This work has been supported by ARPA, through ONR grant N00014-91-J-1989.

Contents

Chapter 1

Overview

The Massively Scalar Compiler is a research compiler being constructed at Rice University.[1] We are mainly concerned with exploring issues in analysis, optimization, and code generation. The primary issues of interest are *effectiveness* and *efficiency*; however, we have also undertaken the task of *explaining* our work.

1.1 Effectiveness

1.2 Efficiency

1.3 Explanation

[1] The term "massively scalar" is due to a slip of the tongue by Cliff Click at PLDI in 1992.

Chapter 2

The Front End

2.1 Fortran to Iloc

2.2 C to Iloc

2.3 Other Possibilities

Chapter 3

The Optimizer

Many passes. Highly factored – separation of concerns

Optimizations and transformations

3.1 Optimizations

3.1.1 Eliminating Dead Code

unreachable code versus useless code

[5]

3.1.2 Eliminating Redundant Code

Partial Redundancy Elimination

[8, 6]

3.1.3 Hash-Based Value Numbering

3.1.4 Strength Reduction

[4, 10]

3.1.5 Compile-Time Evaluation

Constant Propagation

[9]

Global Peephole Optimization

3.2 Tranformations

These don't really improve the code directly; instead, they rewrite the code so that the optimizations will be more effective. We like the term *enabling transformations.*

3.2.1 Loop Peeling

3.2.2 Assertions

3.2.3 Reassociation of Loop Invariants

[3]

3.2.4 Partition-Based Value Numbering

[1]

3.2.5 Other Transformations

expanding operations

Chapter 4

Code Generation

4.1 Instruction Scheduling

basic blocks, software pipelining, other things

4.2 Register Allocation

[2]

4.3 Clean

removes empty basic blocks (effectively, jumps to jumps).

Need to run last because register allocation (coalescing) can remove all the instructions from a basic block

Chapter 5

Translating to C

for testing, debugging, measurements

Bibliography

[1] Bowen Alpern, Mark N. Wegman, and F. Kenneth Zadeck. Detecting equality of variables in programs. In *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, pages 1–11, San Diego, California, January 1988.

[2] Preston Briggs. *Register Allocation via Graph Coloring.* PhD thesis, Rice University, April 1992.

[3] Preston Briggs and Keith D. Cooper. Effective partial redundancy elimination. Submitted to Sigplan PLDI., November 1993.

[4] John Cocke and Ken Kennedy. An algorithm for reduction of operator strength. *Communications of the ACM*, 20(11), November 1977.

[5] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[6] Karl-Heinz Drechsler and Manfred P. Stadel. A solution to a problem with Morel and Renvoise's "Global optimization by suppression of partial redundancies". *ACM Transactions on Programming Languages and Systems*, 10(4):635–640, October 1988.

[7] David R. Hanson. Fast allocation and deallocation of memory based on object lifetimes. *Software – Practice and Experience*, 20(1):5–12, January 1990.

[8] Etienne Morel and Claude Renvoise. Global optimization by suppression of partial redundancies. *Communications of the ACM*, 22(2):96–103, February 1979.

[9] Mark N. Wegman and F. Kenneth Zadeck. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems*, 13(2):181–210, April 1991.

[10] Michael Wolfe. Beyond induction variables. *SIGPLAN Notices*, 27(7):162–174, July 1992. *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation.*

Appendix A

The Shared Code

"shared.h" 8a ≡
```
#ifndef SHARED
#define SHARED 1
#include <stdio.h>
⟨Defining ABORT 8b⟩
⟨Standard type definitions 9a, ... ⟩
⟨Timing interface specifications 10a⟩
⟨Arena interface specifications 11b⟩
⟨Sparse set definitions 15b⟩
⟨Vector set definitions 13a⟩
⟨Internal representation 18c⟩
#endif /* SHARED */
◇
```

A.1 Exit versus Abort

⟨Defining ABORT 8b⟩ ≡
```
#ifdef RUNTIME_CHECK
#define ABORT abort()
#else
#define ABORT exit(-1)
#endif
◇
```
Macro referenced in scrap 8a.

"shared.index" 8c ≡
```
ABORT
◇
```
File defined by scraps 8c, 11a, 12c, 15a, 18b, 23df, 25c, 29c, 30c, 32d, 34c, 36b, 38b, 39b, 40bd, 41ace, 43b, 45b, 46bd, 47b, 48b, 49b, 50b, 51b, 52b, 54b, 56bdf, 57bd, 58ac, 59ace, 60aceg, 61bdf, 62bdf, 63bd, 64bdf, 65bdf, 66aceg, 67bd, 68ac.

A.2 Standard Compiler Types

As an aid to portability, our group uses a number of local names for supposedly standard types. The hope is that simple modifications to these typedefs will ease porting to a variety of machines.

As a matter of personal preference, we have adopted the convention that all type names will begin with an uppercase letter. Therefore, we must rename some common types. We start with the major types inherited from `stdio.h`.

⟨Standard type definitions 9a⟩ ≡

```
typedef FILE *File;
```

◇

Macro defined by scraps 9abcde.
Macro referenced in scrap 8a.

We also give new names for some of the standard compiler types. Note that some care should be exercised with the definition of `Char`. There's often a question about whether the the basic `char` is signed or not. For the `xlc` compiler on the RS/6000, it is apparently unsigned by default.

⟨Standard type definitions 9b⟩ ≡

```
typedef char Char;
typedef double Double;
typedef void Void;
```

◇

Macro defined by scraps 9abcde.
Macro referenced in scrap 8a.

The type `Int` is supposed to be the fastest "natural" integer on the machine. We use it for local variables in a routine. The other variations are to be used when it is important to specify the amount of space required for storage – typically in the definition of arrays and structures.

⟨Standard type definitions 9c⟩ ≡

```
typedef int Int;
typedef signed char Int1;
typedef int Int2;
typedef long Int4;
```

◇

Macro defined by scraps 9abcde.
Macro referenced in scrap 8a.

Similar considerations apply to the unsigned integers.

⟨Standard type definitions 9d⟩ ≡

```
typedef unsigned int Unsigned_Int;
typedef unsigned char Unsigned_Int1;
typedef unsigned int Unsigned_Int2;
typedef unsigned long Unsigned_Int4;
```

◇

Macro defined by scraps 9abcde.
Macro referenced in scrap 8a.

We define an explicit type for boolean results. Note that the definition of `Boolean` is not really related to the definition of `TRUE` and `FALSE`. We use this approach since the compiler would otherwise allocate 4 bytes for every boolean. Thus, the use of `Boolean`, `TRUE` and `FALSE` is almost exclusively for documentation purposes.

⟨Standard type definitions 9e⟩ ≡

```
#undef FALSE
#undef TRUE
enum bool {
  FALSE = 0,
  TRUE = 1
};

typedef Unsigned_Int1 Boolean;
```

◇

Macro defined by scraps 9abcde.
Macro referenced in scrap 8a.

A.3 Timing

This is a simple timing package that supports measurement of time requirements for various parts of the optimizer.

⟨Timing interface specifications 10a⟩ ≡
    ⟨Timer types 10b⟩
    ⟨Timer variables 10e⟩
    ⟨Timer prototypes 10c, ... ⟩
    ◇
Macro referenced in scrap 8a.

The `Timer` type is used to declare timer variables. It's declared in terms of `struct tms`, a hidden system declaration.

⟨Timer types 10b⟩ ≡
```
typedef struct tms *Timer;
```
◇
Macro referenced in scrap 10a.

The routine `Time_Start` allocates, initializes, and returns a timer variable.

⟨Timer prototypes 10c⟩ ≡
```
Timer Time_Start(Void);
```
◇
Macro defined by scraps 10cd.
Macro referenced in scrap 10a.

The routine `Time_Dump` takes a timer variable and prints the elapsed user time (since initialization or a call to `Time_Dump`) to `stderr`. The elapsed time is represented as a floating-point number of seconds. The timer variable is updated at every call, so it may be used to time several intervals in sequence.

The parameter `format_string` specifies a format to be used internally by `fprintf`. Overall, a typical usage of the package is structured like this:

```
Timer time = Time_Start();
⟨First pass⟩
Time_Dump(time, "Pass 1 required %1.2f seconds\n");
⟨Second pass⟩
Time_Dump(time, "Pass 2 required %1.2f seconds\n");
free(time);
```

Be sure to note the call to `free(time)`.

⟨Timer prototypes 10d⟩ ≡
```
Void Time_Dump(Timer timer_var, Char *format_string);
```
◇
Macro defined by scraps 10cd.
Macro referenced in scrap 10a.

The global variable `time_print` is used to control `Time_Dump`. If `time_print` is set to `TRUE`, `Time_Dump` will print timing information to `stderr`; otherwise, `Time_Dump` will not print.

By default, `time_print` is `FALSE`. Note that the flag may be toggled at any time without unexpect side effects; it serves only to control `Time_Dump`.

⟨Timer variables 10e⟩ ≡
```
extern Boolean time_print;
```
◇
Macro referenced in scrap 10a.

"shared.index" 11a ≡

```
Timer Time_Start Time_Dump time_print
◇
```

File defined by scraps 8c, 11a, 12c, 15a, 18b, 23df, 25c, 29c, 30c, 32d, 34c, 36b, 38b, 39b, 40bd, 41ace, 43b, 45b, 46bd, 47b, 48b, 49b, 50b, 51b, 52b, 54b, 56bdf, 57bd, 58ac, 59ace, 60aceg, 61bdf, 62bdf, 63bd, 64bdf, 65bdf, 66aceg, 67bd, 68ac.

A.4 Memory Management

Arenas provide a memory management scheme which allows quick allocation and deallocation of blocks which have the same lifetime. The idea was described by Hanson [7].

⟨Arena interface specifications 11b⟩ ≡

⟨Arena types 11c⟩
⟨Arena prototypes 11d, ... ⟩
◇

Macro referenced in scrap 8a.

An arena is a pool of memory from which blocks are allocated as needed. Allocation usually takes small-constant time (the arena package must occasionally allocate more memory internally). When the blocks are no longer needed, the entire arena can be deallocated with a single constant-time operation.

Arenas have no inherent size restrictions. However, if there is not enough system memory to satisfy a request for memory, the arena package will print a message to `stderr` and abort.

The `Arena` type is used to handle arenas. Users cannot access the fields of the arena structure. We declare the `Arena` type using a typedef to `union arena`. The type `union arena` is defined only in the implementation of arenas.

⟨Arena types 11c⟩ ≡

```
typedef union arena *Arena;
◇
```

Macro referenced in scrap 11b.

We have a pair of routines used for creating and destroying arenas. `Arena_Create` returns an arena for use with the various allocation routines. When we're finished with a particular arena, we can release all the storage associated with it (quickly) by passing it to `Arena_Destroy`. After an arena has been destroyed, it may not be used to allocate more space; instead, a new arena must be created.

⟨Arena prototypes 11d⟩ ≡

```
Arena Arena_Create(Void);
Void Arena_Destroy(Arena);
◇
```

Macro defined by scraps 11de, 12ab.
Macro referenced in scrap 11b.

There are two main routines used to allocate storage from an arena. The first, `Arena_GetMem`, returns a pointer to an uninitialized chunk of memory at least `num_bytes` long. The second, `Arena_GetMemClear`, returns a pointer to a zeroed chunk of memory at least `num_bytes` long. Obviously, `Arena_GetMemClear` requires O(`num_bytes`) time to clear the chunk, though a fast system routine is used.

⟨Arena prototypes 11e⟩ ≡

```
Void *Arena_GetMem(Arena, Unsigned_Int num_bytes);
Void *Arena_GetMemClear(Arena, Unsigned_Int num_bytes);
◇
```

Macro defined by scraps 11de, 12ab.
Macro referenced in scrap 11b.

Sometimes it's handy to allocate and free space using a stack-like discipline. In fact, `alloca` is provided in many (but not all) C implementations for just such a purpose. To help promote portability of our code, we use our own implementation.

The idea is to mark an arena, allocate a bunch of stuff, then release all the stuff in one fell swoop. To be useful though, we allow some allocations, a mark, some allocations, a mark, a free, more allocations, etc., with the arena (conceptually) expanding and contracting like a stack.

⟨Arena prototypes 12a⟩ ≡

```
Void Arena_Mark(Arena);
Void Arena_Release(Arena);
```

◇

Macro defined by scraps 11de, 12ab.
Macro referenced in scrap 11b.

To help collect experimental data on the space requirements of each optimization, we have included code to moniter the memory usage of our arenas. Invoking `Arena_DumpStats` forces a short printout of the usage information to `stderr`. Note that the usage information is only accumulated if `Arena.c` has been compiled with the symbol `RUNTIME_CHECK` defined (typically, only in the `-g` version of the shared archive).

⟨Arena prototypes 12b⟩ ≡

```
Void Arena_DumpStats(Void);
```

◇

Macro defined by scraps 11de, 12ab.
Macro referenced in scrap 11b.

"shared.index" 12c ≡

```
Arena Arena_Create Arena_Destroy Arena_GetMem Arena_GetMemClear
Arena_Mark Arena_Release Arena_DumpStats
```

◇

File defined by scraps 8c, 11a, 12c, 15a, 18b, 23df, 25c, 29c, 30c, 32d, 34c, 36b, 38b, 39b, 40bd, 41ace, 43b, 45b, 46bd, 47b, 48b, 49b, 50b, 51b, 52b, 54b, 56bdf, 57bd, 58ac, 59ace, 60aceg, 61bdf, 62bdf, 63bd, 64bdf, 65bdf, 66aceg, 67bd, 68ac.

A.5 Vector Sets

A vector set is a set abstraction which can handle sets of integers over the universe $\{0, 1, 2, \ldots, N-1\}$, where N is specified when the set is created. Vector sets are manipulated using the type `VectorSet`. Vector sets use 1 bit per element in the universe and the universe is limited to 64K elements. The time complexities of the various operations are:

Create	O(N)
Clear	O(N)
Insert	O(1)
Delete	O(1)
Member	O(1)
Size	O(N)
ChooseMember	O(N)
Copy	O(N)
Equal	O(N)
Union	O(N)
Intersect	O(N)
Difference	O(N)
Complement	O(N)
ForAll	O(N)

where N is the size of the universe. Note that the big advantage of these sets are their relative space efficiency and the fact that they take advantage of the machine's bit-wise operations for computation of the boolean operations.

⟨Vector set definitions 13a⟩ ≡
⟨Vector types 13b⟩
⟨Vector macros 13c⟩
⟨Vector prototypes 13d, … ⟩
◇

Macro referenced in scrap 8a.

We've got to expose the inner structure of a set so that the iterator described in the next section can work properly. Nevertheless, we don't want to catch anyone taking advantage of this.

⟨Vector types 13b⟩ ≡
```
struct vector_set {
  Unsigned_Int4 universe_size;
  Unsigned_Int4 word_count;
  Unsigned_Int4 word[1];
};

typedef struct vector_set *VectorSet;
```
◇

Macro referenced in scrap 13a.

For convenience, we provide a macro for iterating over all the members in a set. It's not particularly efficient, requiring time proportional to the size of the universe. The idea is to use it like a for statement, saying something like `VectorSet_ForAll(i, live_set){ A }`. This would repeatedly execute the statements in `A`, each time with a different value for `i`. Of course, the values of `i` would be chosen from the members of `live_set`. For correct behavior, be sure not to modify `live_set` inside the loop.

As a side effect of implementation, the iterator always visits the members of a set in order.

⟨Vector macros 13c⟩ ≡
```
#define VectorSet_ForAll(x, set)                  \
  for (x = 0; x < (set)->universe_size; x++)     \
    if (VectorSet_Member((set), x))
```
◇

Macro referenced in scrap 13a.

When creating a set, we need to specify the size of the universe (maximum 64K elements) and the arena where it should be allocated. When the set is created, it is initially cleared; though `VectorSet_Clear` can be used at other times.

⟨Vector prototypes 13d⟩ ≡
```
VectorSet VectorSet_Create(Arena, Unsigned_Int universe_size);
Void VectorSet_Clear(VectorSet);
```
◇

Macro defined by scraps 13de, 14abcdef.
Macro referenced in scrap 13a.

We also provide routines to insert and delete members from a set. It is not an error to add a duplicate member nor is it an error to attempt to remove a number that is not a member of the set. On the other hand, it is an error to add or delete a number outside the set's universe. `VectorSet_Member` allows us to check if a number is a member of a set. `VectorSet_Size` returns the current cardinality of the set; that is, the number of members in the set.

⟨Vector prototypes 13e⟩ ≡
```
Void VectorSet_Insert(VectorSet, Unsigned_Int);
Void VectorSet_Delete(VectorSet, Unsigned_Int);
Boolean VectorSet_Member(VectorSet, Unsigned_Int);
Unsigned_Int VectorSet_Size(VectorSet);
```
◇

Macro defined by scraps 13de, 14abcdef.
Macro referenced in scrap 13a.

`VectorSet_ChooseMember` returns an arbitrary member of a set. The set is not modified. The set should have at least one member.

⟨Vector prototypes 14a⟩ ≡
```
Unsigned_Int VectorSet_ChooseMember(VectorSet);
◇
```
Macro defined by scraps 13de, 14abcdef.
Macro referenced in scrap 13a.

These routines are used to copy and compare sets. The sets should have the same size universe.

⟨Vector prototypes 14b⟩ ≡
```
Void VectorSet_Copy(VectorSet dst, VectorSet src);
Boolean VectorSet_Equal(VectorSet, VectorSet);
◇
```
Macro defined by scraps 13de, 14abcdef.
Macro referenced in scrap 13a.

These four routines provide common bitwise operations between two sets. All use a three-address form, so that `VectorSet_Difference`, for example, computes $x \leftarrow y \cap \overline{z}$. Aliasing of the operands *is* allowed; that is, you can directly compute $x \leftarrow x \cap \overline{y}$ or $x \leftarrow y \cap \overline{x}$. In each case, the sets should have the same size universe. Note that `VectorSet_Imply(x, y, z)` computes $x \leftarrow \overline{y} \cup z$.

⟨Vector prototypes 14c⟩ ≡
```
Void VectorSet_Union(VectorSet x, VectorSet y, VectorSet z);
Void VectorSet_Intersect(VectorSet x, VectorSet y, VectorSet z);
Void VectorSet_Difference(VectorSet x, VectorSet y, VectorSet z);
Void VectorSet_Imply(VectorSet x, VectorSet y, VectorSet z);
◇
```
Macro defined by scraps 13de, 14abcdef.
Macro referenced in scrap 13a.

We also provide a routine that computes the commonly used sequence $w \leftarrow (x \cap \overline{y}) \cup z$. Again, aliasing is allowed and all the sets should have the same sized universe.

⟨Vector prototypes 14d⟩ ≡
```
Void VectorSet_DifferenceUnion(VectorSet w, VectorSet x,
                               VectorSet y, VectorSet z);
◇
```
Macro defined by scraps 13de, 14abcdef.
Macro referenced in scrap 13a.

`VectorSet_Complement` computes the complement of a set; that is, $x \leftarrow \overline{y}$. Again, we allow the arguments to be aliased, computing $x \leftarrow \overline{x}$

⟨Vector prototypes 14e⟩ ≡
```
Void VectorSet_Complement(VectorSet x, VectorSet y);
◇
```
Macro defined by scraps 13de, 14abcdef.
Macro referenced in scrap 13a.

When debugging, it's sometimes desirable to display the contents of a set. `VectorSet_Dump` prints out all the members of a set to `stderr`.

⟨Vector prototypes 14f⟩ ≡
```
Void VectorSet_Dump(VectorSet);
◇
```
Macro defined by scraps 13de, 14abcdef.
Macro referenced in scrap 13a.

"shared.index" 15a ≡

```
VectorSet VectorSet_ForAll VectorSet_Create VectorSet_Clear
VectorSet_Delete VectorSet_Member VectorSet_Size VectorSet_Insert
VectorSet_ChooseMember VectorSet_Copy VectorSet_Equal
VectorSet_Union VectorSet_Intersect VectorSet_Difference VectorSet_Imply
VectorSet_DifferenceUnion VectorSet_Complement VectorSet_Dump
```

◇

File defined by scraps 8c, 11a, 12c, 15a, 18b, 23df, 25c, 29c, 30c, 32d, 34c, 36b, 38b, 39b, 40bd, 41ace, 43b, 45b, 46bd, 47b, 48b, 49b, 50b, 51b, 52b, 54b, 56bdf, 57bd, 58ac, 59ace, 60aceg, 61bdf, 62bdf, 63bd, 64bdf, 65bdf, 66aceg, 67bd, 68ac.

A.6 Sparse Sets

A sparse set is a set abstraction which can handle sets of integers over the universe $\{0, 1, 2, \ldots, N-1\}$, where N is specified when the set is created. Sparse sets are manipulated using the type `SparseSet`, which is returned by `SparseSet_Create`. Sparse sets require 4 bytes per element in the universe and the universe is limited to 64K elements. The time complexities of the various operations are:

Create	O(1)
Clear	O(1)
Insert	O(1)
Delete	O(1)
Member	O(1)
Size	O(1)
ChooseMember	O(1)
Copy	O(n)
Equal	O(n)
Union	O(n)
Intersect	O(n)
Difference	O(n)
Complement	O(N)
Sort	O($n \log n$)
ForAll	O(n)

where n is the current set size and N is the size of the universe. The implementation is described in Briggs' thesis [2, pages 69–70].

⟨Sparse set definitions 15b⟩ ≡
  ⟨Sparse types 16a⟩
  ⟨Sparse macros 16b⟩
  ⟨Sparse prototypes 16c, … ⟩
  ◇

Macro referenced in scrap 8a.

We've got to expose some of the inner structure of a set so that the iterator described in the next section can work properly. Nevertheless, we don't want to catch anyone taking advantage of this.

⟨Sparse types 16a⟩ ≡

```
struct sparse_set_node {
  Unsigned_Int2 stack_element;
  Unsigned_Int2 stack_index;
};

struct sparse_set {
  Unsigned_Int2 universe_size;
  Unsigned_Int2 member_count;
  struct sparse_set_node node[1];
};

typedef struct sparse_set *SparseSet;
◇
```

Macro referenced in scrap 15b.

One of the advantages of the sparse set is that we can quickly iterate over all the members. The macro below is provided for convenience. The idea is to use it like a **for** statement, saying something like **SparseSet_ForAll(i, live_set) { A }**. This would repeatedly execute the statements in **A**, each time with a different value for **i**. Of course, the values of **i** would be chosen from the members of **live_set**. For correct behavior, be sure not to modify **live_set** inside the loop.

⟨Sparse macros 16b⟩ ≡

```
#define SparseSet_ForAll(x, set)                       \
  for (x = (set)->member_count;                        \
       x && (x = (set)->node[x-1].stack_element, 1);   \
       x = (set)->node[x].stack_index)
◇
```

Macro referenced in scrap 15b.

When creating a set, we need to specify the size of the universe (a maximum of 64K elements) and the arena where it should be allocated. When the set is created, it is initially cleared; though **SparseSet_Clear** can be used at other times.

⟨Sparse prototypes 16c⟩ ≡

```
SparseSet SparseSet_Create(Arena, Unsigned_Int universe_size);
Void SparseSet_Clear(SparseSet);
◇
```

Macro defined by scraps 16cd, 17abcde, 18a.
Macro referenced in scrap 15b.

We also provide routines to insert and delete members from a set. It is not an error to add a duplicate member nor is it an error to attempt to remove a number that is not a member of the set. However, **x** should always be within the universe. **SparseSet_Member** allows us to check if a number is a member of a set. **SparseSet_Size** returns the current cardinality of the set; that is, the number of members.

⟨Sparse prototypes 16d⟩ ≡

```
Void SparseSet_Insert(SparseSet, Unsigned_Int x);
Void SparseSet_Delete(SparseSet, Unsigned_Int x);
Boolean SparseSet_Member(SparseSet, Unsigned_Int x);
Unsigned_Int SparseSet_Size(SparseSet);
◇
```

Macro defined by scraps 16cd, 17abcde, 18a.
Macro referenced in scrap 15b.

SparseSet_ChooseMember returns an arbitrary member of a set. The set should not be empty. The set is not modified.

⟨Sparse prototypes 17a⟩ ≡

```
Unsigned_Int SparseSet_ChooseMember(SparseSet);
```

◇

Macro defined by scraps 16cd, 17abcde, 18a.
Macro referenced in scrap 15b.

We also provide routines to copy one set to another and to compare two sets for equality. The sets should have the same size universe.

⟨Sparse prototypes 17b⟩ ≡

```
Boolean SparseSet_Equal(SparseSet, SparseSet);
Void SparseSet_Copy(SparseSet dst, SparseSet src);
```

◇

Macro defined by scraps 16cd, 17abcde, 18a.
Macro referenced in scrap 15b.

We provide three routines, computing common operations between two sets. All use a three-address form, so that `SparseSet_Difference`, for example, computes $x \leftarrow y \cap \overline{z}$. Aliasing of the operands *is* allowed; that is, you can directly compute $x \leftarrow x \cap \overline{y}$ or $x \leftarrow y \cap \overline{x}$. In all cases, the sets should have the same size universe.

⟨Sparse prototypes 17c⟩ ≡

```
Void SparseSet_Union(SparseSet x, SparseSet y, SparseSet z);
Void SparseSet_Intersect(SparseSet x, SparseSet y, SparseSet z);
Void SparseSet_Difference(SparseSet x, SparseSet y, SparseSet z);
```

◇

Macro defined by scraps 16cd, 17abcde, 18a.
Macro referenced in scrap 15b.

`SparseSet_Complement` computes the complement of a set; that is, $x \leftarrow \overline{y}$. Again, we allow the operands to be aliased, computing $x \leftarrow \overline{y}$. Note that this routine is fairly expensive, requiring $O(N)$ time, where N is the size of the universe. Fortunately, we can often avoid the need for this routine by using the routine `SparseSet_Difference`.

⟨Sparse prototypes 17d⟩ ≡

```
Void SparseSet_Complement(SparseSet x, SparseSet y);
```

◇

Macro defined by scraps 16cd, 17abcde, 18a.
Macro referenced in scrap 15b.

Though sets are supposed to be unordered collections of objects, we occasionally find it useful to ensure that the iterator macro will visit all the members in order. `SparseSet_Sort` rearranges things so that members will be visited lowest number first. Of course, other set operations can immediately spoil the ordering.

⟨Sparse prototypes 17e⟩ ≡

```
Void SparseSet_Sort(SparseSet);
```

◇

Macro defined by scraps 16cd, 17abcde, 18a.
Macro referenced in scrap 15b.

When debugging, it's sometimes desirable to display the contents of a set. `SparseSet_Dump` prints out all the members of a set to `stderr`. For readability, the routine first orders the elements, so the user needn't bother. Sometimes, we also want to run the output of some of the code through awk, which has a limit to the number of fields that can appear on a line. In this case, the output from `SparseSet_Dump` will cause awk to fail. Thus, we have another debugging routine, `SparseSet_Dump_Pretty`, which adds newlines to limit the number of members appearing on a single line.

⟨Sparse prototypes 18a⟩ ≡

```
Void SparseSet_Dump(SparseSet);
Void SparseSet_Dump_Pretty(SparseSet);
◇
```

Macro defined by scraps 16cd, 17abcde, 18a.
Macro referenced in scrap 15b.

"shared.index" 18b ≡

```
SparseSet SparseSet_ForAll SparseSet_Create SparseSet_Clear
SparseSet_Insert SparseSet_Delete SparseSet_Member SparseSet_Size
SparseSet_ChooseMember SparseSet_Equal SparseSet_Copy
SparseSet_Union SparseSet_Intersect SparseSet_Difference
SparseSet_Complement SparseSet_Sort SparseSet_Dump SparseSet_Dump_Pretty
◇
```

File defined by scraps 8c, 11a, 12c, 15a, 18b, 23df, 25c, 29c, 30c, 32d, 34c, 36b, 38b, 39b, 40bd, 41ace, 43b, 45b, 46bd, 47b, 48b, 49b, 50b, 51b, 52b, 54b, 56bdf, 57bd, 58ac, 59ace, 60aceg, 61bdf, 62bdf, 63bd, 64bdf, 65bdf, 66aceg, 67bd, 68ac.

A.7 The Internal Representation

The control-flow graph is basically a directed graph, with nodes (basic blocks) and edges. The directed graph is also a flow graph, in that there is a `start_block` from which we can reach all other blocks. To support certain algorithms that work on the reverse control-flow graph, we also have an `end_block`. There are distinct data structures for both blocks and edges, giving us convenient places to hang information associated with the control-flow graph (*e.g.*, the results of data-flow analysis).

Classically, basic blocks are a sequence of instructions, a set of incoming edges, and a set of outgoing edges. We also keep a set of the labels associated a block and various other flags to support analysis. Further, both edges and basic blocks support an "extension" structure, allowing each pass of the optimizer to keep private information associated with particular blocks or edges (of course, C++ would support this ability directly; but we're not working with C++).

In ILOC, *instructions* have a particular meaning: An instruction is a set of operations that may be executed in parallel. An *operation* is the specification of a single computation, rather like an assembly-language instruction. An operation consists of an opcode and zero or more arguments. The arguments of an operation are either register numbers or expressions.

The registers are simply unsigned numbers, chosen arbitrarily from an infinite set. In practice, we use (and encourage) a dense numbering, beginning at zero. We only allocate 2 bytes for each number, so we are limited to 64K registers.

Expressions (including integer constants, floating-point constants, labels, and so forth) are all stored uniquely in a table and referred to via an index into the table. Again, we only allocate 2 bytes for each index, so we are limited to 64K expressions.

⟨Internal representation 18c⟩ ≡

```
⟨Expressions 35a, ... ⟩
⟨Opcodes 32b, ... ⟩
⟨Operations 31a, ... ⟩
⟨Instructions 29d, ... ⟩
⟨Blocks and Edges 25d, ... ⟩
⟨Control-flow graph 19a, ... ⟩
⟨Parser-code definitions 70a, ... ⟩
◇
```

Macro referenced in scrap 8a.

A.7.1 The Control-Flow Graph

At the beginning of an optimization pass, the graph is constructed by calling `Block_Init` with the name of an ILOC file. The argument may be `NULL`, in which case `stdin` is used for the source file. The file is scanned and parsed and the expression table and the control-flow graph are constructed. If any errors are detected, they are reported and execution is terminated. After constructing the graph, any unreachable blocks are deleted. Finally, the vectors `preorder_block_list` and `postorder_block_list` are created (see below).

⟨Control-flow graph 19a⟩ ≡
```
Void Block_Init(Char *file_name);
◇
```
Macro defined by scraps 19abcde, 20abcd, 21abc, 22abc, 23abce, 24abcd, 25ab.
Macro referenced in scrap 18c.

Two global flags control construction of the graph. The flag `insert_landing_pads` controls insertion of empty basic blocks before the head of each loop in the control-flow graph. Such empty blocks are called *landing pads*, since they provide a unique place to insert loop-invariant expressions before a loop.

Similarly, the flag `insert_edge_splits` controls the splitting of edges. If true, edges from a block x and to a block y, where x has more than one successor and y has more than one predecessor are split into two edges with a new, empty block z. Thus, $\vec{xy}$ becomes $\vec{xz}$ and $\vec{zy}$.

Both of these flags are false by default.

⟨Control-flow graph 19b⟩ ≡
```
extern Boolean insert_landing_pads;
extern Boolean insert_edge_splits;
◇
```
Macro defined by scraps 19abcde, 20abcd, 21abc, 22abc, 23abce, 24abcd, 25ab.
Macro referenced in scrap 18c.

A third flag is used to indicate that ILOC comments are to be kept or discarded. If `keep_comments` is true (the default case), comments are preserved in a compact form when the control-flow graph is constructed. If false, the comments will be discarded as the ILOC is read.

The main use of the comments is to aid in debugging. The disadvantages of preserving them are slightly increased time and space requirements.

⟨Control-flow graph 19c⟩ ≡
```
extern Boolean keep_comments;
◇
```
Macro defined by scraps 19abcde, 20abcd, 21abc, 22abc, 23abce, 24abcd, 25ab.
Macro referenced in scrap 18c.

I need to say more about these, but later. Tim.

⟨Control-flow graph 19d⟩ ≡
```
extern Boolean replace_source_line_ref_with_line_count;
extern Boolean replace_source_line_ref_with_character_count;
◇
```
Macro defined by scraps 19abcde, 20abcd, 21abc, 22abc, 23abce, 24abcd, 25ab.
Macro referenced in scrap 18c.

When the graph is constructed, a unique, empty start block is added to the graph, with edges leading from it to any entry points (designated by a `FRAME` instruction). Similarly, a unique, empty end block is inserted, with edges leading to it from all returns and halts. The new blocks are pointed to by the global variables `start_block` and `end_block`. Notice that this approach handles the case of a Fortran subroutine with multiple entry points as well as the case where several ILOC routines are packed into a single file.

⟨Control-flow graph 19e⟩ ≡
```
extern Block *start_block;
extern Block *end_block;
◇
```
Macro defined by scraps 19abcde, 20abcd, 21abc, 22abc, 23abce, 24abcd, 25ab.
Macro referenced in scrap 18c.

Memory for the control-flow graph is allocated in an arena called `block_memory_bank`. Additions to the graph might also be allocated in the same arena, but the arena should not be destroyed until the graph has been written out.

⟨Control-flow graph 20a⟩ ≡
```
extern Arena block_memory_bank;
◇
```
Macro defined by scraps 19abcde, 20abcd, 21abc, 22abc, 23abce, 24abcd, 25ab.
Macro referenced in scrap 18c.

While creating the control-flow graph, we keep track of the number of edges and blocks in the graph.

⟨Control-flow graph 20b⟩ ≡
```
extern Unsigned_Int block_count;
extern Unsigned_Int edge_count;
◇
```
Macro defined by scraps 19abcde, 20abcd, 21abc, 22abc, 23abce, 24abcd, 25ab.
Macro referenced in scrap 18c.

At the same time, we also record the largest register used in the code, thus enabling the simple creation of sets of integers for data-flow analysis. Note that we are allowed to create larger register numbers when convenient; however, `register_count` should be updated.

We also keep track of the total number of tags seen in the code in the variable `tag_count`

⟨Control-flow graph 20c⟩ ≡
```
extern Unsigned_Int register_count; /* largest register plus one */
  extern Unsigned_Int tag_count;/* total number of tags*/
◇
```
Macro defined by scraps 19abcde, 20abcd, 21abc, 22abc, 23abce, 24abcd, 25ab.
Macro referenced in scrap 18c.

When the control-flow graph is created (via invocation of `Block_Init`), two vectors of pointers to blocks are created: `preorder_block_list` and `postorder_block_list`. Each vector is `block_count+2` long, with `NULL` values at each end of the vector. The remaining entries are pointers to basic blocks. The following invariant is true:

$$\texttt{preorder\_block\_list[1]} = \texttt{postorder\_block\_list[block\_count]} = \texttt{start\_block}$$

⟨Control-flow graph 20d⟩ ≡
```
extern Block **preorder_block_list;
extern Block **postorder_block_list;
◇
```
Macro defined by scraps 19abcde, 20abcd, 21abc, 22abc, 23abce, 24abcd, 25ab.
Macro referenced in scrap 18c.

By examining the blocks in the order defined by `preorder_block_list` or `postorder_block_list`, we can traverse the control-flow graph in any of several commonly useful orders. We define macros to simplify the actual code. In each case `b` is a pointer to `Block`. A typical use might be:

```
ForAllBlocks_Preorder(block) {
   Inst *inst;
   Block_ForAllInsts(inst, block) {
      ...
   }
}
```

Note that the block lists are invalidated if the control-flow graph is modified (edges or blocks added or deleted). To recreate the orderings after modification, call `Block_Order` (defined below).

⟨Control-flow graph 21a⟩ ≡
```
#define ForAllBlocks_Preorder(b)                          \
   for (b=preorder_block_list[1];                         \
        b;                                                \
        b = preorder_block_list[b->preorder_index+1])

#define ForAllBlocks_rPreorder(b)                         \
   for (b=preorder_block_list[block_count];               \
        b;                                                \
        b = preorder_block_list[b->preorder_index-1])

#define ForAllBlocks_Postorder(b)                         \
   for (b=postorder_block_list[1];                        \
        b;                                                \
        b = postorder_block_list[b->postorder_index+1])

#define ForAllBlocks_rPostorder(b)                        \
   for (b=postorder_block_list[block_count];              \
        b;                                                \
        b = postorder_block_list[b->postorder_index-1])
```
◇

Macro defined by scraps 19abcde, 20abcd, 21abc, 22abc, 23abce, 24abcd, 25ab.
Macro referenced in scrap 18c.

We also define a macro for use when traversing the control-flow graph in an arbitrary order.

⟨Control-flow graph 21b⟩ ≡
```
#define ForAllBlocks(b) ForAllBlocks_rPostorder(b)
```
◇

Macro defined by scraps 19abcde, 20abcd, 21abc, 22abc, 23abce, 24abcd, 25ab.
Macro referenced in scrap 18c.

Finally, we define four macros for use when traversing separate routines in the graph. The implementation and handling of these data structures is somewhat cumbersome, and we will refrain from giving the full-blown details here. In short, we will define two arrays which are almost identical to the `preorder_block_list` and `postorder_block_list` arrays. The biggest difference is that `end_block` will appear multiple times (once in each routine's list of blocks) in the preoder array, and `start_block` will appear multiple times in the postorder array. Each routine's list of blocks in these arrays will be separated by a zero, and either end of the array will have a zero, as well.

⟨Control-flow graph 21c⟩ ≡
```
extern Block **preorder_subgraph_list;
extern Block **postorder_subgraph_list;
```
◇

Macro defined by scraps 19abcde, 20abcd, 21abc, 22abc, 23abce, 24abcd, 25ab.
Macro referenced in scrap 18c.

We will define another array, indexed by a block's preorder index, which will hold indices into the above arrays of the start and end of that routine's blocks. Thus, most of this array will be empty – the only fields that will be filled in are the entries corresponding to a block containing a `FRAME` statement. Since the start and end of both the preorder and postorder arrays will be the same, we only need one array to hold the information.

⟨Control-flow graph 22a⟩ ≡

```
typedef struct subgraph_points {
    Unsigned_Int front;
    Unsigned_Int back;
} Subgraph_Points;
extern Subgraph_Points **subgraph_points;
◇
```

Macro defined by scraps 19abcde, 20abcd, 21abc, 22abc, 23abce, 24abcd, 25ab.
Macro referenced in scrap 18c.

The following macros were designed to work the same way as the various `ForAll` macros introduced previously. These macros, however, require that the user also include an extra unsigned integer variable, which gets used as the loop counter.

⟨Control-flow graph 22b⟩ ≡

```
#define ForSubgraph_Preorder(i, b, start)                              \
   for (i = subgraph_points[start->preorder_index]->front, b=start; \
        b;                                                           \
        b = preorder_subgraph_list[++i])

#define ForSubgraph_rPreorder(i, b, start)                             \
   for (i = subgraph_points[start->preorder_index]->back,           \
        b=preorder_subgraph_list[subgraph_points[start->preorder_index]->back];\
        b;                                                           \
        b = preorder_subgraph_list[--i])

#define ForSubgraph_Postorder(i, b, start)                             \
   for (i = subgraph_points[start->preorder_index]->front,          \
        b=postorder_subgraph_list[subgraph_points[start->preorder_index]->front]; \
        b;                                                           \
        b = postorder_subgraph_list[++i])

#define ForSubgraph_rPostorder(i, b, start)                            \
   for (i = subgraph_points[start->preorder_index]->back,           \
        b=postorder_subgraph_list[subgraph_points[start->preorder_index]->back]; \
        b;                                                           \
        b = postorder_subgraph_list[--i])
◇
```

Macro defined by scraps 19abcde, 20abcd, 21abc, 22abc, 23abce, 24abcd, 25ab.
Macro referenced in scrap 18c.

Sometimes new blocks are created and added to the control-flow graph, invalidating the orderings. To recreate them from the new graph, invoke `Block_Order`. Note that the correct behavior of `Block_Order` depends on the correct value of `block_count`. Therefore, when adding new blocks to the control-flow graph, be sure to increment `block_count` before calling `Block_Order`.

⟨Control-flow graph 22c⟩ ≡

```
Void Block_Order(Void);
◇
```

Macro defined by scraps 19abcde, 20abcd, 21abc, 22abc, 23abce, 24abcd, 25ab.
Macro referenced in scrap 18c.

After optimization is complete, the control-flow graph may be written back to a file in ILOC form by calling `Block_Put_All` and passing in the output file. Note that correct behavior of `Block_Put_All` depends on correct definition of `preorder_block_list`; therefore, be sure to call `Block_Order` if any new blocks have been created.

⟨Control-flow graph 23a⟩ ≡

```
Void Block_Put_All(File);
◇
```

Macro defined by scraps 19abcde, 20abcd, 21abc, 22abc, 23abce, 24abcd, 25ab.
Macro referenced in scrap 18c.

The global variable `print_all_operations` controls output of the ILOC. If false, `Block_Put_All` will print only those operations whose `critical` flag has been set. If `print_all_operations` is true (the default case), all operations will be printed, regardless of the value of their critical flag. This technique simplifies deletion of operations and is particularly useful during dead code elimination.

⟨Control-flow graph 23b⟩ ≡

```
extern Boolean print_all_operations;
◇
```

Macro defined by scraps 19abcde, 20abcd, 21abc, 22abc, 23abce, 24abcd, 25ab.
Macro referenced in scrap 18c.

We have also defined a pair of routines for dumping out debugging information. `Block_Dump_All` dumps out all the blocks in the control-flow graph; `Block_Edge_Dump` dumps all the edges. These are primarily intended for development, perhaps for use from within dbx. Each routine takes as an argument a pointer to a function for printing the (block or edge) extension. If the pointer is `NULL`, the extension is not printed. Note that both routines use `ForAllBlocks`; therefore, `Block_Order` must be called if any new blocks have been added to the graph.

⟨Control-flow graph 23c⟩ ≡

```
Void Block_Dump_All(Void (*extension_printer)(Block *), Boolean);
Void Block_Edge_Dump(Void (*extension_printer)(Edge *));
◇
```

Macro defined by scraps 19abcde, 20abcd, 21abc, 22abc, 23abce, 24abcd, 25ab.
Macro referenced in scrap 18c.

"shared.index" 23d ≡

```
Block_Init insert_landing_pads insert_edge_splits keep_comments
start_block end_block block_memory_bank block_count edge_count
register_count preorder_block_list postorder_block_list
ForAllBlocks_Preorder ForAllBlocks_rPreorder
ForAllBlocks_Postorder ForAllBlocks_rPostorder
ForAllBlocks Block_Order Block_Put_All print_all_operations
Block_Dump_All Block_Edge_Dump
◇
```

File defined by scraps 8c, 11a, 12c, 15a, 18b, 23df, 25c, 29c, 30c, 32d, 34c, 36b, 38b, 39b, 40bd, 41ace, 43b, 45b, 46bd, 47b, 48b, 49b, 50b, 51b, 52b, 54b, 56bdf, 57bd, 58ac, 59ace, 60aceg, 61bdf, 62bdf, 63bd, 64bdf, 65bdf, 66aceg, 67bd, 68ac.

Finally, we have found when debugging that it is useful to be able to find the name or names of a basic block. To facilitate this, we include the following call, which takes a `Block` pointer and prints out the list of labels associated with that block to `stderr`. Also, we will return a pointer to the first label in the block so that debuggers will print that value.

⟨Control-flow graph 23e⟩ ≡

```
Char *Block_Print_Name(Block *block);
◇
```

Macro defined by scraps 19abcde, 20abcd, 21abc, 22abc, 23abce, 24abcd, 25ab.
Macro referenced in scrap 18c.

"shared.index" 23f ≡

```
Block_Print_Name
◇
```

File defined by scraps 8c, 11a, 12c, 15a, 18b, 23df, 25c, 29c, 30c, 32d, 34c, 36b, 38b, 39b, 40bd, 41ace, 43b, 45b, 46bd, 47b, 48b, 49b, 50b, 51b, 52b, 54b, 56bdf, 57bd, 58ac, 59ace, 60aceg, 61bdf, 62bdf, 63bd, 64bdf, 65bdf, 66aceg, 67bd, 68ac.

Dominators and Dominance Frontiers

The notions of dominance and dominance frontiers are quite important for optimization. Since efficient computation of dominators is somewhat difficult to implement, we have included it in our shared archive.

`Dominator_CalcDom` calculates dominators and dominance frontiers. It takes an arena as its only parameter. The arena is used to hold dominator information attached to each block.

⟨Control-flow graph 24a⟩ ≡

```
Void Dominator_CalcDom(Arena);
◇
```

Macro defined by scraps 19abcde, 20abcd, 21abc, 22abc, 23abce, 24abcd, 25ab.
Macro referenced in scrap 18c.

`Dominator_CalcPostDom` calculates post dominators and reverse dominance frontiers. It also takes an arena as its only parameter.

⟨Control-flow graph 24b⟩ ≡

```
Void Dominator_CalcPostDom(Arena);
◇
```

Macro defined by scraps 19abcde, 20abcd, 21abc, 22abc, 23abce, 24abcd, 25ab.
Macro referenced in scrap 18c.

Before calling `Dominator_CalcPostDom`, we need to make sure that any infinite loops are somehow connected to `end_block`; otherwise, the results of optimization will be very difficult to debug. By calling the routine `Block_Find_Infinite_Loops`, we cause extra edges to be inserted in the flow graph, leading to `end_block` from blocks that would not otherwise reach `end_block` (*e.g.*, blocks that are part of an infinite loop).

⟨Control-flow graph 24c⟩ ≡

```
Void Block_Find_Infinite_Loops(Void);
◇
```

Macro defined by scraps 19abcde, 20abcd, 21abc, 22abc, 23abce, 24abcd, 25ab.
Macro referenced in scrap 18c.

The dominator information attached to each block is declared here.

⟨Control-flow graph 24d⟩ ≡

```
struct dom_node {
  Block *parent; /* a pointer to the parent in the dominator tree,
                    NULL if the root of the tree */
  Block_List_Node *children; /* a list of the children in the dominator
                    tree */
  Block_List_Node *frontier; /* a list of the blocks in this block's
                    dominance frontier */
};
◇
```

Macro defined by scraps 19abcde, 20abcd, 21abc, 22abc, 23abce, 24abcd, 25ab.
Macro referenced in scrap 18c.

We define a pair of macros to simplify traversal of the list of children and dominance frontiers. Note that the `b` parameter is a pointer to a `Block_List_Node`, not a pointer to a `Block`. Furthermore, `node` is a pointer to a `DomNode`, not a block. An example of typical usage is given here:

```
Dominator_ForChildren(bn, block->dom_node) {
  Inst *inst;
  Block *child = bn->block;
  Block_ForAllInsts(inst, child) {
    ...
  }
}
```

For each macro, we specify `block->dom_node` to visit blocks in the forward dominator tree; similarly, we specify `block->post_dom_node` to visit blocks in the reverse dominator tree.

⟨Control-flow graph 25a⟩ ≡

```
#define Dominator_ForChildren(b, node) \
   for (b = (node)->children;          \
        b;                             \
        b = b->next)

#define Dominator_ForFrontier(b, node) \
   for (b = (node)->frontier;          \
        b;                             \
        b = b->next)
```

◇

Macro defined by scraps 19abcde, 20abcd, 21abc, 22abc, 23abce, 24abcd, 25ab.
Macro referenced in scrap 18c.

We provide a routine to dump the dominance information. If forward information is desired, `start_block` and `TRUE` should be passed as arguments; otherwise, pass `end_block` and `FALSE`.

⟨Control-flow graph 25b⟩ ≡

```
Void Dominator_Dump(Block *root, Boolean forward_flag);
```

◇

Macro defined by scraps 19abcde, 20abcd, 21abc, 22abc, 23abce, 24abcd, 25ab.
Macro referenced in scrap 18c.

"shared.index" 25c ≡

```
Dominator_CalcDom Dominator_CalcPostDom Block_Find_Infinite_Loops
Dominator_ForChildren Dominator_ForFrontier Dominator_Dump
```

◇

File defined by scraps 8c, 11a, 12c, 15a, 18b, 23df, 25c, 29c, 30c, 32d, 34c, 36b, 38b, 39b, 40bd, 41ace, 43b, 45b, 46bd, 47b, 48b, 49b, 50b, 51b, 52b, 54b, 56bdf, 57bd, 58ac, 59ace, 60aceg, 61bdf, 62bdf, 63bd, 64bdf, 65bdf, 66aceg, 67bd, 68ac.

A.7.2 Blocks and Edges

An abstract graph has nodes (or vertices) and edges. Our control-flow graph uses basic blocks for nodes and has explicit directed edges (instead of simply connecting blocks directly to other blocks). Maintaining separate structures for edges allows us to hang information on edges directly.

⟨Blocks and Edges 25d⟩ ≡

```
typedef struct block Block;
typedef struct edge Edge;
```

◇

Macro defined by scraps 25de, 26abcde, 27, 28abc, 29ab.
Macro referenced in scrap 18c.

To allow a readable representation of an arbitrary graph, each block has one or more symbolic labels. Each label is represented internally as an `Expr`; each block has a list of labels. While it would be possible to clean up all extraneous labels, leaving only one label per block, we find it useful to preserve all labels as an aid to debugging.

⟨Blocks and Edges 25e⟩ ≡

```
typedef struct label_node Label;

struct label_node {
  Label *next;
  Expr label;
};
```

◇

Macro defined by scraps 25de, 26abcde, 27, 28abc, 29ab.
Macro referenced in scrap 18c.

Given a symbolic label, a user can get the corresponding string with the `Label_Get_String` function.

⟨Blocks and Edges 26a⟩ ≡
```
Char *Label_Get_String(Expr label);
```
◇

Macro defined by scraps 25de, 26abcde, 27, 28abc, 29ab.
Macro referenced in scrap 18c.

The function `Label_Get_Destination` returns the unique block associated with a label.

⟨Blocks and Edges 26b⟩ ≡
```
Block *Label_Get_Destination(Expr label);
```
◇

Macro defined by scraps 25de, 26abcde, 27, 28abc, 29ab.
Macro referenced in scrap 18c.

Of course, the user has to have a way to know that the `Expr` value is really a `Label`. We provide the following `Boolean` function, which returns `TRUE` if the `Expr` is a `Label`.

⟨Blocks and Edges 26c⟩ ≡
```
Boolean Expr_Is_Block_Label(Expr expression);
```
◇

Macro defined by scraps 25de, 26abcde, 27, 28abc, 29ab.
Macro referenced in scrap 18c.

We provide entry points (described in the file `Dominators.w`) to compute the forward and reverse dominator tree and dominance frontiers. The tree and frontiers are attached directly to blocks using `DomNode` structures.

⟨Blocks and Edges 26d⟩ ≡
```
typedef struct dom_node DomNode;
```
◇

Macro defined by scraps 25de, 26abcde, 27, 28abc, 29ab.
Macro referenced in scrap 18c.

Since each pass of the optimizer may need to hang different information (and different types of information) on blocks and edges, each block and edge has (potentially) an extension, defined locally to each pass.

⟨Blocks and Edges 26e⟩ ≡
```
typedef struct block_extension Block_Extension;
typedef struct edge_extension Edge_Extension;
```
◇

Macro defined by scraps 25de, 26abcde, 27, 28abc, 29ab.
Macro referenced in scrap 18c.

This idea is perhaps non-intuitive. Basically, blocks and edges have a field reserved for a pointer to an extension. The actual extension will be a `struct block_extension` (or `struct edge_extension`), but the actual `struct` has not yet been specified. In each optimizer pass that requires a block extension, we'll have to provide an appropriate definition. For example:

```
struct block_extension {
   VectorSet live_in;
   VectorSet live_out;
}
```

A typical reference to this extension might look like:

```
if (member(block->block_extension->live_in, reg)) {
   ...
}
```

A basic block has many components. As a part of the control-flow graph, it has a set of labels, a set of incoming edges, and a set of outgoing edges, where each set is implemented as a singly-linked list. It also contains a sequence of instructions, implemented as a circularly- and doubly-linked list.

As part of the information associated with the control-flow graph, we maintain vectors of blocks, recording the results of preorder and postorder traversals. The fields `preorder_index` and `postorder_index` give each block's position in these vectors (see the definition of `preorder_block_list`).

Most of the fields shown below are initialized when the control-flow graph is created. Exceptions are `dom_node`, `post_dom_node`, `visited`, and `block_extension`.

The fields `dom_node` and `post_dom_node` are initially set to `NULL`; they are filled in more meaningfully using calls to `Dominator_CalcDom` and `Dominator_CalcPostDom`.

The value of `visited` is used to control recursive walks around the control-flow graph. It is always assumed to be undefined and must always be initialized before use (perhaps using the `ForAllBlocks` macro).

The `block_extension` field is also initialized to `NULL`. It should be filled in when the block extension is created (assuming a block extension is defined).

Finally, we need to spend a little while explaining how the instructions in a block are organized. There are always at least two instructions: a `NOP` and a control-flow instruction (*i.e.,* a jump, branch, return, or halt). The instructions are organized in a circularly-linked list, with the `NOP` acting as the header node. The list is doubly-linked for quick insertion and deletion. A block's `inst` field points at the `NOP` instruction. The first "real" instruction of the block is the `NOP`'s successor. The last instruction in a block (the control-flow instruction) is the `NOP`'s predecessor.

⟨Blocks and Edges 27⟩ ≡

```
struct block {
  Label *labels;                  /* list of labels for the block */
  Inst *inst;                     /* points to head of circularly-linked list of
                                     instructions in the block */
  Edge *pred;                     /* list of incoming edges */
  Edge *succ;                     /* list of outgoing edges */
  Unsigned_Int2 preorder_index;   /* this block's position in preorder_block_list */
  Unsigned_Int2 postorder_index;  /* this block's position in postorder_block_list */
  Unsigned_Int2 pred_count;       /* number of predecessors in the graph */
  DomNode *dom_node;              /* forward dominator information */
  DomNode *post_dom_node;         /* reverse dominator info */
  Boolean visited;                /* occasionally used during traversals of the graph */
  Unsigned_Int2 descendant_count; /* number of descendants in the depth-first
                                     spanning tree (used to find loops) */
  Block_Extension *block_extension;
};
```
◇

Macro defined by scraps 25de, 26abcde, 27, 28abc, 29ab.
Macro referenced in scrap 18c.

Each edge has pointers to its source and target blocks. Furthermore, each edge is part of two sets: a set of incoming edges for the target block and a set of outgoing edges for the source block. These sets are implemented as singly-linked lists and each edge contains pointers to the next element in each list. We also find it convenient to mark *back edges* explicitly.

All of the fields shown below are initialized when the control-flow graph is created. The exception is `extension`, which must be declared and initialized as part of each optimizer pass.

⟨Blocks and Edges 28a⟩ ≡

```
struct edge {
  Edge *next_pred;      /* rest of the predecessor list for the target block */
  Edge *next_succ;      /* rest of the successor list for the source block */
  Block *pred;          /* source of the edge */
  Block *succ;          /* target of the edge */
  Boolean back_edge;    /* TRUE if this is a back edge */
  Edge_Extension *edge_extension;
};
```

◇

Macro defined by scraps 25de, 26abcde, 27, 28abc, 29ab.
Macro referenced in scrap 18c.

Given the definition of `Block`, we often find it convenient to keep sets of blocks in the form of a list.

⟨Blocks and Edges 28b⟩ ≡

```
typedef struct block_node Block_List_Node;
struct block_node {
  Block_List_Node *next;
  Block *block;
};
```

◇

Macro defined by scraps 25de, 26abcde, 27, 28abc, 29ab.
Macro referenced in scrap 18c.

We define a pair of macros to allow easy traversal of all the instructions in a block. `Block_ForAllInsts` visits the instructions from first to last; `Block_ForAllInstsReverse` visits them in reverse order. Each is intended to look like a `for` statement. In each case, `inst` should be a pointer to an `Inst` and `block` should be a pointer to an `Block`. Typically, the macros will be used like this:

```
Block_ForAllInsts(inst, block) {
   Operation **op;
   Inst_ForAllOperations(op, inst) {
      ...
   }
}
```

Be careful not to modify `inst` or `block` within the loop.

⟨Blocks and Edges 28c⟩ ≡

```
#define Block_ForAllInsts(instruction, block)          \
   for (instruction = (block)->inst->next_inst;        \
        instruction != (block)->inst;                  \
        instruction = instruction->next_inst)

#define Block_ForAllInstsReverse(instruction, block)   \
   for (instruction = (block)->inst->prev_inst;        \
        instruction != (block)->inst;                  \
        instruction = instruction->prev_inst)
```

◇

Macro defined by scraps 25de, 26abcde, 27, 28abc, 29ab.
Macro referenced in scrap 18c.

We also define a pair of macros to allow simple traversal over the edges entering and exiting a block. The parameter `edge` is a pointer to an `Edge` and `block` is a pointer to a `Block`. As usual, care should be exercised to avoid modifying `edge` within the loop.

⟨Blocks and Edges 29a⟩ ≡

```
#define Block_ForAllPreds(edge, block)          \
   for (edge = (block)->pred;                   \
        edge;                                   \
        edge = edge->next_pred)

#define Block_ForAllSuccs(edge, block)          \
   for (edge = (block)->succ;                   \
        edge;                                   \
        edge = edge->next_succ)
```

◇

Macro defined by scraps 25de, 26abcde, 27, 28abc, 29ab.
Macro referenced in scrap 18c.

We provide the routine `Block_Dump` as an aid to debugging. It prints a `block`'s data to `stderr`. The `extension_printer` parameter is a pointer to a function that prints out the `Block_Extension` data; if `NULL`, then it is ignored.

⟨Blocks and Edges 29b⟩ ≡

```
Void Block_Dump(Block *block, Void (*extension_printer)(Block *), Boolean);
```

◇

Macro defined by scraps 25de, 26abcde, 27, 28abc, 29ab.
Macro referenced in scrap 18c.

"shared.index" 29c ≡

```
Label Label_Get_Destination Block_Extension Edge_Extension
Block_List_Node Block_ForAllInsts Block_ForAllInstsReverse
Block_ForAllPreds Block_ForAllSuccs Block_Dump
```

◇

File defined by scraps 8c, 11a, 12c, 15a, 18b, 23df, 25c, 29c, 30c, 32d, 34c, 36b, 38b, 39b, 40bd, 41ace, 43b, 45b, 46bd, 47b, 48b, 49b, 50b, 51b, 52b, 54b, 56bdf, 57bd, 58ac, 59ace, 60aceg, 61bdf, 62bdf, 63bd, 64bdf, 65bdf, 66aceg, 67bd, 68ac.

A.7.3 Instructions

To allow code generation for machines with wide-instruction words, each ILOC instruction may have one or more operations. Pointers to the operations are stored in a vector `operations` that is allocated to the correct length. The vector of operation pointers is terminated with a `NULL` entry. If there is a control-flow operation (e.g., a jump) in the vector, it will appear first.

Each instruction is part of a basic block. The instructions in a basic block are kept in a doubly-linked list, where each instruction has pointers to the next and previous instructions.

⟨Instructions 29d⟩ ≡

```
typedef struct inst_node Inst;

struct inst_node {
  Inst *prev_inst;
  Inst *next_inst;
  Operation *operations[1];
};
```

◇

Macro defined by scraps 29d, 30ab.
Macro referenced in scrap 18c.

When a new instruction is created, space must be allocated for all the fields and all the operations. We hide the details in a small macro definition. The `arena` argument specifies where the memory will come from; `ops` specifies the number of operations in the instruction.

Note that the declaration of `operations` allows room for the `NULL` terminator.

⟨Instructions 30a⟩ ≡
```
#define Inst_Allocate(arena, ops) \
  Arena_GetMem(arena, sizeof(Inst) + ops * sizeof(Operation *))
```
◇

Macro defined by scraps 29d, 30ab.
Macro referenced in scrap 18c.

We also define a macro to allow easy traversal of all the operations in an instruction. It is intended to look like a `for` statement. In each case, `op` should be a pointer to an `Operation *` and `inst` should be a pointer to an `Inst`. Typically, the macros will be used like this:

```
Inst_ForAllOperations(op, inst) {
   Operation *operation = *op;
   if (opcode_specs[operation->opcode].details & EXPR) {
      ...
   }
}
```

Be careful not to modify `op` or `inst` within the loop (though `*op` and `**op` are fair game). Note well that `op` does not point at an operation; instead, `op` points at a pointer to an operation.

⟨Instructions 30b⟩ ≡
```
#define Inst_ForAllOperations(op, inst) \
   for (op = &(inst)->operations[0]; *op; op++)
```
◇

Macro defined by scraps 29d, 30ab.
Macro referenced in scrap 18c.

"`shared.index`" 30c ≡
```
Inst_Allocate Inst_ForAllOperations
```
◇

File defined by scraps 8c, 11a, 12c, 15a, 18b, 23df, 25c, 29c, 30c, 32d, 34c, 36b, 38b, 39b, 40bd, 41ace, 43b, 45b, 46bd, 47b, 48b, 49b, 50b, 51b, 52b, 54b, 56bdf, 57bd, 58ac, 59ace, 60aceg, 61bdf, 62bdf, 63bd, 64bdf, 65bdf, 66aceg, 67bd, 68ac.

A.7.4 Operations

An ILOC operation is represented internally as an `Operation`. The `opcode` field is used to refer to the opcode definition (mostly useful for accessing the `details` field).

Notice the difference between the `constants` field here and the `consts` field in the definition of `Opcode`. The `consts` field specifies the number of *expected* constants; the `constants` field specifies the number of constants actually found.

Note also how the `referenced` and `defined` fields are specified. The number of referenced (or *used*) registers equals `referenced-constants`. Similarly, the number of defined registers equal `defined-referenced`. The methodology allows simple traversal of the different classes of registers. For example, to examine the referenced registers, we simply examine `arguments[i]`, where `constants <= i < referenced`.

The `arguments` vector holds expression indices (referring to any constants in the operation) and register numbers. It is not always 3 elements long; instead, we allocate each operation enough space to hold all its arguments. Some operations may have 1 or 2 lists of expressions (*e.g.*, the lists of referenced and modified variables associated with a `JSRl`). These lists are handled by placing expression indices at the end of the `arguments` vector, with each list terminated by a 0 entry. Thus, an operation of the form:

```
iJSRl zip r10 r11 => r12 [@foo @bar] [@quux]
```

would have an `arguments` vector with 10 entries: the expression index of `zip`, the numbers 10, 11, and 12, the expression indices of `foo` and `bar`, a zero, the expression index of `quux` and a final zero. The respective values for `constants`, `referenced`, and `defined` would be 1, 3, and 4.

⟨Operations 31a⟩ ≡

```
typedef struct {
  Opcode_Names opcode;
  Unsigned_Int2 comment;         /* index of optional comment */
  Expr source_line_ref;          /* reference back to the original program source */
  Unsigned_Int1 constants;       /* the number of constant actually found */
  Unsigned_Int1 referenced;      /* constants + references */
  Unsigned_Int1 defined;         /* constants + references + defs */
  Boolean critical;              /* true if operation is not dead */
  Unsigned_Int2 arguments[3];    /* an array of arguments (expressions and register numbers) */
} Operation;
```

◇

Macro defined by scraps 31abc, 32ac.
Macro referenced in scrap 18c.

When a new operation is created, perhaps as the result of optimization, space must be allocated for all the fields and arguments. We hide the details in a small macro definition. The `arena` argument specifies where the memory will come from; `args` specifies the number of arguments the operations will have.

Note the term `args-3`. We subtract 3 to compensate for the number of elements already provided for in the declaration of `arguments`.

⟨Operations 31b⟩ ≡

```
#define Operation_Allocate(arena, args) \
  Arena_GetMem(arena, sizeof(Operation) + (args-3)*sizeof(Unsigned_Int2))
```

◇

Macro defined by scraps 31abc, 32ac.
Macro referenced in scrap 18c.

We also define three macros used to visit the arguments of an operation. Each is intended to look like a `for` statement. In each case, `op` should be a pointer to an `Operation` and `p` should be a pointer to an `Unsigned_Int2`. Typically, the macros will be used like this:

```
Operation_ForAllDefs(p, op) {
   Unsigned_Int reg = *p;
   if (member(reg, live_in)) {
      ...
   }
}
```

Be careful not to modify `op` or `p` within the loop (though `*p` is fair game).

⟨Operations 31c⟩ ≡

```
#define Operation_ForAllConstants(p, op)        \
   for (p = &(op)->arguments[0];                \
        p < &(op)->arguments[(op)->constants];  \
        p++)

#define Operation_ForAllUses(p, op)             \
   for (p = &(op)->arguments[(op)->constants];  \
        p < &(op)->arguments[(op)->referenced]; \
        p++)

#define Operation_ForAllDefs(p, op)             \
   for (p = &(op)->arguments[(op)->referenced]; \
        p < &(op)->arguments[(op)->defined];    \
        p++)
```

◇

Macro defined by scraps 31abc, 32ac.
Macro referenced in scrap 18c.

We also have two macros we use to visit the tag related aguements of an operation which behave much the same way. It is first necessary to declare an external routine `Operation_Second_List_Start` which gives us a pointer to the list of defined tags.

⟨Operations 32a⟩ ≡

```
#define Operation_ForAllRefTags(p, op)                        \
    for(p = (opcode_specs[(op)->opcode].details & USES_TAGS) ? \
                &(op)->arguments[(op)->defined] :          \
                NULL;                                                  \
        p && *p != 0;                                      \
        p++)

extern Unsigned_Int2 *Operation_Second_List_Start(Operation *);

#define Operation_ForAllDefTags(p, op)                            \
    for(p = (opcode_specs[(op)->opcode].details & DEFINES_TAGS) ? \
                Operation_Second_List_Start((op)) :                   \
                NULL;                                                     \
        p && *p != 0;                                         \
        p++)
```

◇

Macro defined by scraps 31abc, 32ac.
Macro referenced in scrap 18c.

Some optimizer passes need to determine the type of each register. To support this, we specify the type of each defined register.

⟨Opcodes 32b⟩ ≡

```
enum def_types {
  NO_DEFS, INT_DEF, FLOAT_DEF, DOUBLE_DEF,
  COMPLEX_DEF, DCOMPLEX_DEF, MULT_DEFS
};

typedef Unsigned_Int1 Def_Type;
```

◇

Macro defined by scraps 32b, 33abc, 34ab.
Macro referenced in scrap 18c.

Rather than grubbing around in the opcode definitions to find the type of a register, you should use this routine. Note that it only works for the registers *defined* in a particular operation.

⟨Operations 32c⟩ ≡

```
Unsigned_Int Operation_Def_Type(Operation *, Unsigned_Int);
```

◇

Macro defined by scraps 31abc, 32ac.
Macro referenced in scrap 18c.

"shared.index" 32d ≡

```
Operation_Allocate Operation_ForAllConstants
Operation_ForAllUses Operation_ForAllDefs Operation_Def_Type
Def_Type NO_DEFS INT_DEF FLOAT_DEF DOUBLE_DEF COMPLEX_DEF DCOMPLEX_DEF
```

◇

File defined by scraps 8c, 11a, 12c, 15a, 18b, 23df, 25c, 29c, 30c, 32d, 34c, 36b, 38b, 39b, 40bd, 41ace, 43b, 45b, 46bd, 47b, 48b, 49b, 50b, 51b, 52b, 54b, 56bdf, 57bd, 58ac, 59ace, 60aceg, 61bdf, 62bdf, 63bd, 64bdf, 65bdf, 66aceg, 67bd, 68ac.

A.7.5 Opcodes

We have pretty symbolic names for all of our ILOC opcodes; however, there are many of them (over 250), so we define them all in the next appendix.

⟨Opcodes 33a⟩ ≡
```
typedef
  enum {
    ⟨Opcode names 38a, ... ⟩
  } names_of_opcodes;
◇
```
Macro defined by scraps 32b, 33abc, 34ab.
Macro referenced in scrap 18c.

We'll define a new type called `Opcode_Names` to hold the symbolic value whenever desired, though it is basically just a two-byte integer. The obvious typedef to `names_of_opcodes` is less helpful since the compiler allocates 4 bytes for all enumeration types, regardless of the actual values.

⟨Opcodes 33b⟩ ≡
```
typedef Unsigned_Int2 Opcode_Names;
◇
```
Macro defined by scraps 32b, 33abc, 34ab.
Macro referenced in scrap 18c.

The `Opcode` type defines the parameters for each opcode. It is used primarily when inhaling an ILOC file. With the occasional exception of the details field, this structure is largely ignored by the optimizer passes; instead, they tend to refer to the `Operation` data structure defined in the next chapter. Note that the values in these fields are intended to be constant and must not be modified.

⟨Opcodes 33c⟩ ≡
```
typedef struct {
  Char *opcode;                /* the spelling of the opcode */
  Opcode_Names opcode_name;    /* the index that will yield this entry;
                                      used for error checking */
  Int1 consts;                 /* the expected number of constants
                                      (-n indicates at least
                                        n constants expected) */
  Int1 uses;                   /* the expect number of referenced registers */
  Int1 defs;                   /* the expected number of defined registers */
  Int1 lists;                  /* the expected number of expression lists */
  Def_Type def_type;           /* the type of defined registers */
  Unsigned_Int2 details;       /* bit vector specifying interesting details
                                      (see below) */

} Opcode;
◇
```
Macro defined by scraps 32b, 33abc, 34ab.
Macro referenced in scrap 18c.

The `details` field is treated as a bit vector. The idea is to allow quick testing for membership of an opcode in a certain class. For example, if `opcode_specs[x].details & CALL`, then `x` must be a procedure call. Defined values (bits) include:

⟨Opcodes 34a⟩ ≡
```
#define NO_DETAILS      0
#define PSEUDO          1       /* pseudo-ops */
#define CONDITIONAL     (1<<1)  /* conditional branches */
#define CRITICAL        (1<<2)  /* must not be deleted by dead code elimination */
#define DATA            (1<<3)  /* marks data statements */
#define CALL            (1<<4)  /* JSRr and JSRl operations */
#define EXPR            (1<<5)  /* marks "expression" operations */
#define SCALAR          (1<<6)
#define LOAD            (1<<7)  /* operations that load from non-constant memory */
#define STORE           (1<<8)  /* operations that store to memory */
#define COMMUTE         (1<<9)  /* commutative operations */
#define COPY            (1<<10) /* register-to-register copies */
#define USES_TAGS       (1<<11)
#define DEFINES_TAGS    (1<<12)
#define ASSOCIATIVE     (1<<13) /* associative operations */
◇
```
Macro defined by scraps 32b, 33abc, 34ab.
Macro referenced in scrap 18c.

We define a constant table `opcode_specs` containing common information about all the ILOC opcodes; *e.g.*, spelling, expected number of operands, and so forth. The table is used primarily when inhaling a file and initializing the operations; however, it is still useful to have the table accessible to allow reference to the `details` and `def_type` fields.

The size of the table is given by the variable `number_of_opcodes` and the actual values for each entry are given in the shared archive. We also define the number of opcodes using a C macro so that users can declare and initialize arrays using a constant value. Having a copy of the value in an actual variable allows us to access the value from inside a symbolic debugger. In the body of the code where we define `number_of_opcodes`, we simply set it equal to `total_number_of_opcodes`, so there is no danger of having different values defining the number of opcodes in the language.

The opcode table, `opcode_specs`, is indexed by a value of type `Opcode_Names`. Thus, for example, we can use `opcode_specs[iADD].details` to access the details associated with the integer addition operation. The entry for `iADD` looks like:

```
{ "iADD", iADD, 0, 2, 1, 0, INT_DEF, EXPR|COMMUTE }
```

This entry specifies how the opcode is spelled, the opcode name, that it uses no constants, has two register operands, and defines one register. There are no lists expected, the defined register is an integer, and the opcode is a commutative expression.

⟨Opcodes 34b⟩ ≡
```
#define total_number_of_opcodes 333
extern Unsigned_Int number_of_opcodes;
extern Opcode opcode_specs[];
◇
```
Macro defined by scraps 32b, 33abc, 34ab.
Macro referenced in scrap 18c.

"shared.index" 34c ≡
```
NO_DETAILS PSEUDO CONDITIONAL CRITICAL DATA ASSOCIATIVE
CALL EXPR LOAD STORE COMMUTE COPY opcode_specs number_of_opcodes total_number_of_opcodes
◇
```
File defined by scraps 8c, 11a, 12c, 15a, 18b, 23df, 25c, 29c, 30c, 32d, 34c, 36b, 38b, 39b, 40bd, 41ace, 43b, 45b, 46bd, 47b, 48b, 49b, 50b, 51b, 52b, 54b, 56bdf, 57bd, 58ac, 59ace, 60aceg, 61bdf, 62bdf, 63bd, 64bdf, 65bdf, 66aceg, 67bd, 68ac.

A.7.6 Expressions

Immediate values in ILOC operations are allowed to be arbitrarily complex arithmetic expressions. For consistency, we keep all expressions (from simple integers to the more complex messes) in a single table, built up while inhaling the ILOC. All entries in the table, and therefore all expressions, are referred to via an index into the table, of type `Expr`.

⟨Expressions 35a⟩ ≡
```
typedef Unsigned_Int2 Expr;
◇
```
Macro defined by scraps 35abcde, 36a.
Macro referenced in scrap 18c.

During optimization, we sometimes wish to examine an expression to see if it's an integer (especially an important integer like 0 or 1).

⟨Expressions 35b⟩ ≡
```
Boolean Expr_Is_Integer(Expr);
◇
```
Macro defined by scraps 35abcde, 36a.
Macro referenced in scrap 18c.

Occasionally, a pass will want to know the total number of `Expr`s in memory. This is kept internally, but the value can be gotten by calling the following function.

⟨Expressions 35c⟩ ≡
```
Unsigned_Int Expr_Count(Void);
◇
```
Macro defined by scraps 35abcde, 36a.
Macro referenced in scrap 18c.

If an expression is an integer, then it makes sense to examine the integer directly, via `Expr_Get_Integer`. The routine `Expr_Install_Int` provides a way to add new integer expressions to the table (perhaps for the results of constant folding).

⟨Expressions 35d⟩ ≡
```
Int Expr_Get_Integer(Expr);
Expr Expr_Install_Int(Int);
◇
```
Macro defined by scraps 35abcde, 36a.
Macro referenced in scrap 18c.

Even if an expression is not an integer, we will sometimes want to manipulate it directly. The function `Expr_Get_String` returns a pointer to the expression's string representation (this function also works for integers). Note that the string representation should be considered "read only." Don't change it; make a copy of it.

We also provide a corresponding routine `Expr_Install_String` to add a string to the table. It is polite and makes a copy of its argument.

⟨Expressions 35e⟩ ≡
```
Char *Expr_Get_String(Expr);
Expr Expr_Install_String(Char *);
◇
```
Macro defined by scraps 35abcde, 36a.
Macro referenced in scrap 18c.

Lack of interprocedural data-flow information for a particular call is indicated by a "?" in its lists of referenced and modified variables. Since this is an expression, we need a handy way to reference its index. The global variables `question_mark_expr` and `exclamation_point_expr` are initialized when initially constructing the expression table.

⟨Expressions 36a⟩ ≡

```
extern Expr question_mark_expr;
extern Expr exclamation_point_expr;
```

◇

Macro defined by scraps 35abcde, 36a.
Macro referenced in scrap 18c.

"shared.index" 36b ≡

```
Expr_Is_Integer Expr_Get_Integer Expr_Install_Int
Expr_Get_String Expr_Install_String question_mark_expr exclamation_point_expr
```

◇

File defined by scraps 8c, 11a, 12c, 15a, 18b, 23df, 25c, 29c, 30c, 32d, 34c, 36b, 38b, 39b, 40bd, 41ace, 43b, 45b, 46bd, 47b, 48b, 49b, 50b, 51b, 52b, 54b, 56bdf, 57bd, 58ac, 59ace, 60aceg, 61bdf, 62bdf, 63bd, 64bdf, 65bdf, 66aceg, 67bd, 68ac.

Appendix B

The Intermediate Language

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.

B.1 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.

B.2 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*

B.3 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.

⟨Opcode names 38a⟩ ≡

```
NAME, ALIAS,
```
◇

Macro defined by scraps 38a, 39a, 40ace, 41bd, 42, 43a, 45a, 46ac, 47a, 48a, 49a, 50a, 51a, 52a, 54a, 56ace, 57ace, 58bd, 59bdf, 60bdf, 61ace, 62ace, 63ac, 64ace, 65aceg, 66bdf, 67ace, 68b.
Macro referenced in scrap 33a.

`"shared.index"` 38b ≡

```
NAME ALIAS
```
◇

File defined by scraps 8c, 11a, 12c, 15a, 18b, 23df, 25c, 29c, 30c, 32d, 34c, 36b, 38b, 39b, 40bd, 41ace, 43b, 45b, 46bd, 47b, 48b, 49b, 50b, 51b, 52b, 54b, 56bdf, 57bd, 58ac, 59ace, 60aceg, 61bdf, 62bdf, 63bd, 64bdf, 65bdf, 66aceg, 67bd, 68ac.

B.3.1 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.

B.4 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*

`wDATA` *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.

⟨Opcode names 39a⟩ ≡

```
bDATA, wDATA, iDATA, fDATA, dDATA, BYTES,
```

◇

Macro defined by scraps 38a, 39a, 40ace, 41bd, 42, 43a, 45a, 46ac, 47a, 48a, 49a, 50a, 51a, 52a, 54a, 56ace, 57ace, 58bd, 59bdf, 60bdf, 61ace, 62ace, 63ac, 64ace, 65aceg, 66bdf, 67ace, 68b.
Macro referenced in scrap 33a.

"`shared.index`" 39b ≡

```
bDATA wDATA iDATA fDATA dDATA BYTES
```

◇

File defined by scraps 8c, 11a, 12c, 15a, 18b, 23df, 25c, 29c, 30c, 32d, 34c, 36b, 38b, 39b, 40bd, 41ace, 43b, 45b, 46bd, 47b, 48b, 49b, 50b, 51b, 52b, 54b, 56bdf, 57bd, 58ac, 59ace, 60aceg, 61bdf, 62bdf, 63bd, 64bdf, 65bdf, 66aceg, 67bd, 68ac.

B.5 The local stack

The following instructions describe the location of variables allocated on the stack as a function of some offset. We differentiate between scalar and array variables in the same manner as for load and store operations. That is, we append an extra 'S' to the opcode for scalar values.

Scalar values:

`bSSTACK` *offset variable-name*

`wSSTACK` *offset variable-name*

`iSSTACK` *offset variable-name*

`fSSTACK` *offset variable-name*

`dSSTACK` *offset variable-name*

⟨Opcode names 40a⟩ ≡

```
bSSTACK, wSSTACK, iSSTACK, fSSTACK, dSSTACK,
```

◇

Macro defined by scraps 38a, 39a, 40ace, 41bd, 42, 43a, 45a, 46ac, 47a, 48a, 49a, 50a, 51a, 52a, 54a, 56ace, 57ace, 58bd, 59bdf, 60bdf, 61ace, 62ace, 63ac, 64ace, 65aceg, 66bdf, 67ace, 68b.
Macro referenced in scrap 33a.

"shared.index" 40b ≡

```
bSSTACK wSSTACK iSSTACK fSSTACK dSSTACK
```

◇

File defined by scraps 8c, 11a, 12c, 15a, 18b, 23df, 25c, 29c, 30c, 32d, 34c, 36b, 38b, 39b, 40bd, 41ace, 43b, 45b, 46bd, 47b, 48b, 49b, 50b, 51b, 52b, 54b, 56bdf, 57bd, 58ac, 59ace, 60aceg, 61bdf, 62bdf, 63bd, 64bdf, 65bdf, 66aceg, 67bd, 68ac.

Array values:

`bSTACK` *offset variable-name*

`wSTACK` *offset variable-name*

`iSTACK` *offset variable-name*

`fSTACK` *offset variable-name*

`dSTACK` *offset variable-name*

⟨Opcode names 40c⟩ ≡

```
bSTACK, wSTACK, iSTACK, fSTACK, dSTACK,
```

◇

Macro defined by scraps 38a, 39a, 40ace, 41bd, 42, 43a, 45a, 46ac, 47a, 48a, 49a, 50a, 51a, 52a, 54a, 56ace, 57ace, 58bd, 59bdf, 60bdf, 61ace, 62ace, 63ac, 64ace, 65aceg, 66bdf, 67ace, 68b.
Macro referenced in scrap 33a.

"shared.index" 40d ≡

```
bSTACK wSTACK iSTACK fSTACK dSTACK
```

◇

File defined by scraps 8c, 11a, 12c, 15a, 18b, 23df, 25c, 29c, 30c, 32d, 34c, 36b, 38b, 39b, 40bd, 41ace, 43b, 45b, 46bd, 47b, 48b, 49b, 50b, 51b, 52b, 54b, 56bdf, 57bd, 58ac, 59ace, 60aceg, 61bdf, 62bdf, 63bd, 64bdf, 65bdf, 66aceg, 67bd, 68ac.

B.6 Global memory

The following instructions are the global equivalent of the `STACK` instructions shown in the previous section. They describe the location of global memory as a function of some offset. Just as we do for local memory, we differentiate scalar variables from array variables by appending an 'S' to the opcode name.

Scalar values:

`bSGLOBAL` *offset variable-name*

`wSGLOBAL` *offset variable-name*

`iSGLOBAL` *offset variable-name*

`fSGLOBAL` *offset variable-name*

`dSGLOBAL` *offset variable-name*

⟨Opcode names 40e⟩ ≡

```
bSGLOBAL, wSGLOBAL, iSGLOBAL, fSGLOBAL, dSGLOBAL,
```

◇

Macro defined by scraps 38a, 39a, 40ace, 41bd, 42, 43a, 45a, 46ac, 47a, 48a, 49a, 50a, 51a, 52a, 54a, 56ace, 57ace, 58bd, 59bdf, 60bdf, 61ace, 62ace, 63ac, 64ace, 65aceg, 66bdf, 67ace, 68b.
Macro referenced in scrap 33a.

"shared.index" 41a ≡

```
bSGLOBAL wSGLOBAL iSGLOBAL fSGLOBAL dSGLOBAL
```

◇

File defined by scraps 8c, 11a, 12c, 15a, 18b, 23df, 25c, 29c, 30c, 32d, 34c, 36b, 38b, 39b, 40bd, 41ace, 43b, 45b, 46bd, 47b, 48b, 49b, 50b, 51b, 52b, 54b, 56bdf, 57bd, 58ac, 59ace, 60aceg, 61bdf, 62bdf, 63bd, 64bdf, 65bdf, 66aceg, 67bd, 68ac.

Array values:

`bGLOBAL` *offset variable-name*

`wGLOBAL` *offset variable-name*

`iGLOBAL` *offset variable-name*

`fGLOBAL` *offset variable-name*

`dGLOBAL` *offset variable-name*

⟨Opcode names 41b⟩ ≡

```
bGLOBAL, wGLOBAL, iGLOBAL, fGLOBAL, dGLOBAL,
```

◇

Macro defined by scraps 38a, 39a, 40ace, 41bd, 42, 43a, 45a, 46ac, 47a, 48a, 49a, 50a, 51a, 52a, 54a, 56ace, 57ace, 58bd, 59bdf, 60bdf, 61ace, 62ace, 63ac, 64ace, 65aceg, 66bdf, 67ace, 68b.
Macro referenced in scrap 33a.

"shared.index" 41c ≡

```
bGLOBAL wGLOBAL iGLOBAL fGLOBAL dGLOBAL
```

◇

File defined by scraps 8c, 11a, 12c, 15a, 18b, 23df, 25c, 29c, 30c, 32d, 34c, 36b, 38b, 39b, 40bd, 41ace, 43b, 45b, 46bd, 47b, 48b, 49b, 50b, 51b, 52b, 54b, 56bdf, 57bd, 58ac, 59ace, 60aceg, 61bdf, 62bdf, 63bd, 64bdf, 65bdf, 66aceg, 67bd, 68ac.

We also want to include a way to specify the initialization of global variables. To do this, we will use the `XDATA` opcode. It specifies the same information as the `DATA` statement, but tells us that the initialization occurs in global memory, not local memory.

`bXDATA` *integer-expression repetition-expression*

`wXDATA` *integer-expression repetition-expression*

`iXDATA` *integer-expression repetition-expression*

`fXDATA` *single-expression repetition-expression*

`dXDATA` *double-expression repetition-expression*

⟨Opcode names 41d⟩ ≡

```
bXDATA, wXDATA, iXDATA, fXDATA, dXDATA,
```

◇

Macro defined by scraps 38a, 39a, 40ace, 41bd, 42, 43a, 45a, 46ac, 47a, 48a, 49a, 50a, 51a, 52a, 54a, 56ace, 57ace, 58bd, 59bdf, 60bdf, 61ace, 62ace, 63ac, 64ace, 65aceg, 66bdf, 67ace, 68b.
Macro referenced in scrap 33a.

"shared.index" 41e ≡

```
bXDATA wXDATA iXDATA fXDATA dXDATA
```

◇

File defined by scraps 8c, 11a, 12c, 15a, 18b, 23df, 25c, 29c, 30c, 32d, 34c, 36b, 38b, 39b, 40bd, 41ace, 43b, 45b, 46bd, 47b, 48b, 49b, 50b, 51b, 52b, 54b, 56bdf, 57bd, 58ac, 59ace, 60aceg, 61bdf, 62bdf, 63bd, 64bdf, 65bdf, 66aceg, 67bd, 68ac.

Finally, we have to support function pointers in C. To translate these into code that the C compiler will accept (remember that one of our backends is an iloc-to-C translator, we have to be able to insert prototypes for any function name. To do this, our C front end must tell us all of the names of the functions accessed in the file and their type. To carry this information along, we have the **XFUNC** (for *external-function*) statement. The plain **XFUNC** statement corresponds to a function of type **void** in C, while the others correspond to functions returning values of that type.

⟨Opcode names 42⟩ ≡
```
XFUNC, iXFUNC, fXFUNC, dXFUNC,
```
◇

Macro defined by scraps 38a, 39a, 40ace, 41bd, 42, 43a, 45a, 46ac, 47a, 48a, 49a, 50a, 51a, 52a, 54a, 56ace, 57ace, 58bd, 59bdf, 60bdf, 61ace, 62ace, 63ac, 64ace, 65aceg, 66bdf, 67ace, 68b.
Macro referenced in scrap 33a.

B.7 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.

B.8 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.

B.8.1 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.

⟨Opcode names 43a⟩ ≡
```
HALT, NOP,
```
◇

Macro defined by scraps 38a, 39a, 40ace, 41bd, 42, 43a, 45a, 46ac, 47a, 48a, 49a, 50a, 51a, 52a, 54a, 56ace, 57ace, 58bd, 59bdf, 60bdf, 61ace, 62ace, 63ac, 64ace, 65aceg, 66bdf, 67ace, 68b.
Macro referenced in scrap 33a.

"shared.index" 43b ≡
```
HALT NOP
```
◇

File defined by scraps 8c, 11a, 12c, 15a, 18b, 23df, 25c, 29c, 30c, 32d, 34c, 36b, 38b, 39b, 40bd, 41ace, 43b, 45b, 46bd, 47b, 48b, 49b, 50b, 51b, 52b, 54b, 56bdf, 57bd, 58ac, 59ace, 60aceg, 61bdf, 62bdf, 63bd, 64bdf, 65bdf, 66aceg, 67bd, 68ac.

B.8.2 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.

⟨Opcode names 45a⟩ ≡
```
    FRAME,
    JSRr, iJSRr, fJSRr, dJSRr, cJSRr, qJSRr,
    JSRl, iJSRl, fJSRl, dJSRl, cJSRl, qJSRl,
    RTN,  iRTN,  fRTN,  dRTN,  cRTN,  qRTN,
    ◇
```
Macro defined by scraps 38a, 39a, 40ace, 41bd, 42, 43a, 45a, 46ac, 47a, 48a, 49a, 50a, 51a, 52a, 54a, 56ace, 57ace, 58bd, 59bdf, 60bdf, 61ace, 62ace, 63ac, 64ace, 65aceg, 66bdf, 67ace, 68b.
Macro referenced in scrap 33a.

"shared.index" 45b ≡
```
    FRAME
    JSRr iJSRr fJSRr dJSRr cJSRr qJSRr
    JSRl iJSRl fJSRl dJSRl cJSRl qJSRl
    RTN  iRTN  fRTN  dRTN  cRTN  qRTN
    ◇
```
File defined by scraps 8c, 11a, 12c, 15a, 18b, 23df, 25c, 29c, 30c, 32d, 34c, 36b, 38b, 39b, 40bd, 41ace, 43b, 45b, 46bd, 47b, 48b, 49b, 50b, 51b, 52b, 54b, 56bdf, 57bd, 58ac, 59ace, 60aceg, 61bdf, 62bdf, 63bd, 64bdf, 65bdf, 66aceg, 67bd, 68ac.

B.8.3 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.

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.

⟨Opcode names 46a⟩ ≡

```
iLDI, fLDI, dLDI, cLDI, qLDI,
```

◇

Macro defined by scraps 38a, 39a, 40ace, 41bd, 42, 43a, 45a, 46ac, 47a, 48a, 49a, 50a, 51a, 52a, 54a, 56ace, 57ace, 58bd, 59bdf, 60bdf, 61ace, 62ace, 63ac, 64ace, 65aceg, 66bdf, 67ace, 68b.
Macro referenced in scrap 33a.

"shared.index" 46b ≡

```
iLDI fLDI dLDI cLDI qLDI
```

◇

File defined by scraps 8c, 11a, 12c, 15a, 18b, 23df, 25c, 29c, 30c, 32d, 34c, 36b, 38b, 39b, 40bd, 41ace, 43b, 45b, 46bd, 47b, 48b, 49b, 50b, 51b, 52b, 54b, 56bdf, 57bd, 58ac, 59ace, 60aceg, 61bdf, 62bdf, 63bd, 64bdf, 65bdf, 66aceg, 67bd, 68ac.

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.

`wCONor` *tag alignment offset base* `=>` *result* Load an unsigned 2-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.

⟨Opcode names 46c⟩ ≡

```
bCONor, wCONor, iCONor, fCONor, dCONor, cCONor, qCONor,
```

◇

Macro defined by scraps 38a, 39a, 40ace, 41bd, 42, 43a, 45a, 46ac, 47a, 48a, 49a, 50a, 51a, 52a, 54a, 56ace, 57ace, 58bd, 59bdf, 60bdf, 61ace, 62ace, 63ac, 64ace, 65aceg, 66bdf, 67ace, 68b.
Macro referenced in scrap 33a.

"shared.index" 46d ≡

```
bCONor wCONor iCONor fCONor dCONor cCONor qCONor
```

◇

File defined by scraps 8c, 11a, 12c, 15a, 18b, 23df, 25c, 29c, 30c, 32d, 34c, 36b, 38b, 39b, 40bd, 41ace, 43b, 45b, 46bd, 47b, 48b, 49b, 50b, 51b, 52b, 54b, 56bdf, 57bd, 58ac, 59ace, 60aceg, 61bdf, 62bdf, 63bd, 64bdf, 65bdf, 66aceg, 67bd, 68ac.

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.

`wSLDor` *tag alignment offset base* `=>` *result* Load an unsigned 2-byte value from memory. The alignment is included since it might be possible to take advantage of the information to block together operations involving words.

`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.

`wSLDrr` *tag alignment index base* `=>` *result* Load an unsigned 2-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.

⟨Opcode names 47a⟩ ≡
```
bSLDor, wSLDor, iSLDor, fSLDor, dSLDor, cSLDor, qSLDor,
bSLDrr, wSLDrr, iSLDrr, fSLDrr, dSLDrr, cSLDrr, qSLDrr,
◇
```
Macro defined by scraps 38a, 39a, 40ace, 41bd, 42, 43a, 45a, 46ac, 47a, 48a, 49a, 50a, 51a, 52a, 54a, 56ace, 57ace, 58bd, 59bdf, 60bdf, 61ace, 62ace, 63ac, 64ace, 65aceg, 66bdf, 67ace, 68b.
Macro referenced in scrap 33a.

"shared.index" 47b ≡
```
bSLDor wSLDor iSLDor fSLDor dSLDor cSLDor qSLDor
bSLDrr wSLDrr iSLDrr fSLDrr dSLDrr cSLDrr qSLDrr
◇
```
File defined by scraps 8c, 11a, 12c, 15a, 18b, 23df, 25c, 29c, 30c, 32d, 34c, 36b, 38b, 39b, 40bd, 41ace, 43b, 45b, 46bd, 47b, 48b, 49b, 50b, 51b, 52b, 54b, 56bdf, 57bd, 58ac, 59ace, 60aceg, 61bdf, 62bdf, 63bd, 64bdf, 65bdf, 66aceg, 67bd, 68ac.

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.

`wSSTor` *tag alignment offset base value* Store the 2-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.

`wSSTrr` *tag alignment index base value* Store the 2-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.

⟨Opcode names 48a⟩ ≡
```
bSSTor, wSSTor, iSSTor, fSSTor, dSSTor, cSSTor, qSSTor,
bSSTrr, wSSTrr, iSSTrr, fSSTrr, dSSTrr, cSSTrr, qSSTrr,
◇
```
Macro defined by scraps 38a, 39a, 40ace, 41bd, 42, 43a, 45a, 46ac, 47a, 48a, 49a, 50a, 51a, 52a, 54a, 56ace, 57ace, 58bd, 59bdf, 60bdf, 61ace, 62ace, 63ac, 64ace, 65aceg, 66bdf, 67ace, 68b.
Macro referenced in scrap 33a.

"shared.index" 48b ≡
```
bSSTor wSSTor iSSTor fSSTor dSSTor cSSTor qSSTor
bSSTrr wSSTrr iSSTrr fSSTrr dSSTrr cSSTrr qSSTrr
◇
```
File defined by scraps 8c, 11a, 12c, 15a, 18b, 23df, 25c, 29c, 30c, 32d, 34c, 36b, 38b, 39b, 40bd, 41ace, 43b, 45b, 46bd, 47b, 48b, 49b, 50b, 51b, 52b, 54b, 56bdf, 57bd, 58ac, 59ace, 60aceg, 61bdf, 62bdf, 63bd, 64bdf, 65bdf, 66aceg, 67bd, 68ac.

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 `wLDor` *tag alignment offset base* `=>` *result* Load an unsigned 2-byte value from memory. The alignment is included since it might be possible to take advantage of the information to block together operations involving words.

`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.

`wLDrr` *tag alignment index base* `=>` *result* Load an unsigned 2-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.

⟨Opcode names 49a⟩ ≡
```
bLDor, wLDor, iLDor, fLDor, dLDor, cLDor, qLDor,
bLDrr, wLDrr, iLDrr, fLDrr, dLDrr, cLDrr, qLDrr,
```
◇

Macro defined by scraps 38a, 39a, 40ace, 41bd, 42, 43a, 45a, 46ac, 47a, 48a, 49a, 50a, 51a, 52a, 54a, 56ace, 57ace, 58bd, 59bdf, 60bdf, 61ace, 62ace, 63ac, 64ace, 65aceg, 66bdf, 67ace, 68b.
Macro referenced in scrap 33a.

`"shared.index"` 49b ≡
```
bLDor wLDor iLDor fLDor dLDor cLDor qLDor
bLDrr wLDor iLDrr fLDrr dLDrr cLDrr qLDrr
```
◇

File defined by scraps 8c, 11a, 12c, 15a, 18b, 23df, 25c, 29c, 30c, 32d, 34c, 36b, 38b, 39b, 40bd, 41ace, 43b, 45b, 46bd, 47b, 48b, 49b, 50b, 51b, 52b, 54b, 56bdf, 57bd, 58ac, 59ace, 60aceg, 61bdf, 62bdf, 63bd, 64bdf, 65bdf, 66aceg, 67bd, 68ac.

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.

`wSTor` *tag alignment offset base value* Store the 2-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.

`wSTrr` *tag alignment index base value* Store the 2-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.

⟨Opcode names 50a⟩ ≡

```
bSTor, wSTor, iSTor, fSTor, dSTor, cSTor, qSTor,
bSTrr, wSTrr, iSTrr, fSTrr, dSTrr, cSTrr, qSTrr,
```

◇

Macro defined by scraps 38a, 39a, 40ace, 41bd, 42, 43a, 45a, 46ac, 47a, 48a, 49a, 50a, 51a, 52a, 54a, 56ace, 57ace, 58bd, 59bdf, 60bdf, 61ace, 62ace, 63ac, 64ace, 65aceg, 66bdf, 67ace, 68b.
Macro referenced in scrap 33a.

"shared.index" 50b ≡

```
bSTor wSTor iSTor fSTor dSTor cSTor qSTor
bSTrr wSTrr iSTrr fSTrr dSTrr cSTrr qSTrr
```

◇

File defined by scraps 8c, 11a, 12c, 15a, 18b, 23df, 25c, 29c, 30c, 32d, 34c, 36b, 38b, 39b, 40bd, 41ace, 43b, 45b, 46bd, 47b, 48b, 49b, 50b, 51b, 52b, 54b, 56bdf, 57bd, 58ac, 59ace, 60aceg, 61bdf, 62bdf, 63bd, 64bdf, 65bdf, 66aceg, 67bd, 68ac.

B.8.4 Arithmetic

Integer Arithmetic

`iADD` *addend addend* `=>` *sum*

`iSUB` *minuend subtrahend* `=>` *difference* where $difference \leftarrow minuend - subtrahend$.

`iNEG` *operand* `=>` *result*

`iMUL` *multiplicand multiplier* `=>` *product*

`iDIV` *dividend divisor* `=>` *quotient* where $quotient \leftarrow dividend/divisor$.

`iADDI` *immediate addend* `=>` *sum*

`iSUBI` *immediate subtrahend* `=>` *difference* where $difference \leftarrow immediate - subtrahend$.

Single-Precision Arithmetic

`fADD` *addend addend* `=>` *sum*

`fSUB` *minuend subtrahend* `=>` *difference*

`fNEG` *operand* `=>` *result*

`fMUL` *multiplicand multiplier* `=>` *product*

`fDIV` *dividend divisor* `=>` *quotient*

Double-Precision Arithmetic

`dADD` *addend addend* `=>` *sum*

`dSUB` *minuend subtrahend* `=>` *difference*

`dNEG` *operand* `=>` *result*

`dMUL` *multiplicand multiplier* `=>` *product*

`dDIV` *dividend divisor* `=>` *quotient*

Single-Precision Complex Arithmetic

`cADD` *addend addend* `=>` *sum*

`cSUB` *minuend subtrahend* `=>` *difference*

`cNEG` *operand* `=>` *result*

`cMUL` *multiplicand multiplier* `=>` *product*

`cDIV` *dividend divisor* `=>` *quotient*

Double-Precision Complex Arithmetic

`qADD` *addend addend* `=>` *sum*

`qSUB` *minuend subtrahend* `=>` *difference*

`qNEG` *operand* `=>` *result*

`qMUL` *multiplicand multiplier* `=>` *product*

`qDIV` *dividend divisor* `=>` *quotient*

⟨Opcode names 51a⟩ ≡

```
iADD, iSUB, iNEG, iMUL, iDIV, iADDI, iSUBI,
fADD, fSUB, fNEG, fMUL, fDIV,
dADD, dSUB, dNEG, dMUL, dDIV,
cADD, cSUB, cNEG, cMUL, cDIV,
qADD, qSUB, qNEG, qMUL, qDIV,
◇
```

Macro defined by scraps 38a, 39a, 40ace, 41bd, 42, 43a, 45a, 46ac, 47a, 48a, 49a, 50a, 51a, 52a, 54a, 56ace, 57ace, 58bd, 59bdf, 60bdf, 61ace, 62ace, 63ac, 64ace, 65aceg, 66bdf, 67ace, 68b.
Macro referenced in scrap 33a.

"shared.index" 51b ≡

```
iADD iSUB iNEG iMUL iDIV iADDI iSUBI
fADD fSUB fNEG fMUL fDIV
dADD dSUB dNEG dMUL dDIV
cADD cSUB cNEG cMUL cDIV
qADD qSUB qNEG qMUL qDIV
◇
```

File defined by scraps 8c, 11a, 12c, 15a, 18b, 23df, 25c, 29c, 30c, 32d, 34c, 36b, 38b, 39b, 40bd, 41ace, 43b, 45b, 46bd, 47b, 48b, 49b, 50b, 51b, 52b, 54b, 56bdf, 57bd, 58ac, 59ace, 60aceg, 61bdf, 62bdf, 63bd, 64bdf, 65bdf, 66aceg, 67bd, 68ac.

B.8.5 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.

`JMPT` *table 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. We expect to find in the iloc a list of `DATA` statements with the same label as the argument to this operation. The register value will contain an offset into the table of labels. In this way, we can change the destinations of this control-flow operation, for example, to split critical edges, which is not possible with the `JMPr` operation.

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

⟨Opcode names 52a⟩ ≡

```
JMPl, JMPr, JMPT, BR,
◇
```

Macro defined by scraps 38a, 39a, 40ace, 41bd, 42, 43a, 45a, 46ac, 47a, 48a, 49a, 50a, 51a, 52a, 54a, 56ace, 57ace, 58bd, 59bdf, 60bdf, 61ace, 62ace, 63ac, 64ace, 65aceg, 66bdf, 67ace, 68b.
Macro referenced in scrap 33a.

"shared.index" 52b ≡

```
JMPl JMPr JMPT BR
◇
```

File defined by scraps 8c, 11a, 12c, 15a, 18b, 23df, 25c, 29c, 30c, 32d, 34c, 36b, 38b, 39b, 40bd, 41ace, 43b, 45b, 46bd, 47b, 48b, 49b, 50b, 51b, 52b, 54b, 56bdf, 57bd, 58ac, 59ace, 60aceg, 61bdf, 62bdf, 63bd, 64bdf, 65bdf, 66aceg, 67bd, 68ac.

B.8.6 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*

⟨Opcode names 54a⟩ ≡

```
iCMP, fCMP, dCMP, cCMP, qCMP,
EQ, NE, LE, GE, LT, GT,
iCMPeq, iCMPne, iCMPle, iCMPge, iCMPlt, iCMPgt,
fCMPeq, fCMPne, fCMPle, fCMPge, fCMPlt, fCMPgt,
dCMPeq, dCMPne, dCMPle, dCMPge, dCMPlt, dCMPgt,
cCMPeq, cCMPne,
qCMPeq, qCMPne,
BReq, BRne, BRle, BRge, BRlt, BRgt,
◇
```

Macro defined by scraps 38a, 39a, 40ace, 41bd, 42, 43a, 45a, 46ac, 47a, 48a, 49a, 50a, 51a, 52a, 54a, 56ace, 57ace, 58bd, 59bdf, 60bdf, 61ace, 62ace, 63ac, 64ace, 65aceg, 66bdf, 67ace, 68b.
Macro referenced in scrap 33a.

"shared.index" 54b ≡

```
iCMP fCMP dCMP cCMP qCMP
EQ NE LE GE LT GT
iCMPeq iCMPne iCMPle iCMPge iCMPlt iCMPgt
fCMPeq fCMPne fCMPle fCMPge fCMPlt fCMPgt
dCMPeq dCMPne dCMPle dCMPge dCMPlt dCMPgt
cCMPeq cCMPne
qCMPeq qCMPne
BReq BRne BRle BRge BRlt BRgt
◇
```

File defined by scraps 8c, 11a, 12c, 15a, 18b, 23df, 25c, 29c, 30c, 32d, 34c, 36b, 38b, 39b, 40bd, 41ace, 43b, 45b, 46bd, 47b, 48b, 49b, 50b, 51b, 52b, 54b, 56bdf, 57bd, 58ac, 59ace, 60aceg, 61bdf, 62bdf, 63bd, 64bdf, 65bdf, 66aceg, 67bd, 68ac.

B.8.7 Intrinsics

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.

⟨Opcode names 56a⟩ ≡

```
i2i, i2f, i2d, i2c, i2q,
f2i, f2f, f2d, f2c, f2q,
d2i, d2f, d2d, d2c, d2q,
c2i, c2f, c2d, c2c, c2q,
q2i, q2f, q2d, q2c, q2q,
```

◇

Macro defined by scraps 38a, 39a, 40ace, 41bd, 42, 43a, 45a, 46ac, 47a, 48a, 49a, 50a, 51a, 52a, 54a, 56ace, 57ace, 58bd, 59bdf, 60bdf, 61ace, 62ace, 63ac, 64ace, 65aceg, 66bdf, 67ace, 68b.
Macro referenced in scrap 33a.

"shared.index" 56b ≡

```
i2i i2f i2d i2c i2q
f2i f2f f2d f2c f2q
d2i d2f d2d d2c d2q
c2i c2f c2d c2c c2q
q2i q2f q2d q2c q2q
```

◇

File defined by scraps 8c, 11a, 12c, 15a, 18b, 23df, 25c, 29c, 30c, 32d, 34c, 36b, 38b, 39b, 40bd, 41ace, 43b, 45b, 46bd, 47b, 48b, 49b, 50b, 51b, 52b, 54b, 56bdf, 57bd, 58ac, 59ace, 60aceg, 61bdf, 62bdf, 63bd, 64bdf, 65bdf, 66aceg, 67bd, 68ac.

`cCOMPLEX` *real_1 real_2* `=>` *complex_result* Construct a `COMPLEX*8`

`qCOMPLEX` *double_1 double_2* `=>` *dcomplex_result* Construct a `COMPLEX*16`

⟨Opcode names 56c⟩ ≡

```
cCOMPLEX, qCOMPLEX,
```

◇

Macro defined by scraps 38a, 39a, 40ace, 41bd, 42, 43a, 45a, 46ac, 47a, 48a, 49a, 50a, 51a, 52a, 54a, 56ace, 57ace, 58bd, 59bdf, 60bdf, 61ace, 62ace, 63ac, 64ace, 65aceg, 66bdf, 67ace, 68b.
Macro referenced in scrap 33a.

"shared.index" 56d ≡

```
cCOMPLEX qCOMPLEX
```

◇

File defined by scraps 8c, 11a, 12c, 15a, 18b, 23df, 25c, 29c, 30c, 32d, 34c, 36b, 38b, 39b, 40bd, 41ace, 43b, 45b, 46bd, 47b, 48b, 49b, 50b, 51b, 52b, 54b, 56bdf, 57bd, 58ac, 59ace, 60aceg, 61bdf, 62bdf, 63bd, 64bdf, 65bdf, 66aceg, 67bd, 68ac.

Intrinsic Functions

Truncation

`fTRUNC` *real_input* `=>` *real_result* Implements `AINT()`

`dTRUNC` *double_input* `=>` *double_result* Implements `DINT()`

⟨Opcode names 56e⟩ ≡

```
fTRUNC, dTRUNC,
```

◇

Macro defined by scraps 38a, 39a, 40ace, 41bd, 42, 43a, 45a, 46ac, 47a, 48a, 49a, 50a, 51a, 52a, 54a, 56ace, 57ace, 58bd, 59bdf, 60bdf, 61ace, 62ace, 63ac, 64ace, 65aceg, 66bdf, 67ace, 68b.
Macro referenced in scrap 33a.

"shared.index" 56f ≡

```
fTRUNC dTRUNC
```

◇

File defined by scraps 8c, 11a, 12c, 15a, 18b, 23df, 25c, 29c, 30c, 32d, 34c, 36b, 38b, 39b, 40bd, 41ace, 43b, 45b, 46bd, 47b, 48b, 49b, 50b, 51b, 52b, 54b, 56bdf, 57bd, 58ac, 59ace, 60aceg, 61bdf, 62bdf, 63bd, 64bdf, 65bdf, 66aceg, 67bd, 68ac.

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()`

⟨Opcode names 57a⟩ ≡

```
fROUND, dROUND, fNINT, dNINT,
```

◇

Macro defined by scraps 38a, 39a, 40ace, 41bd, 42, 43a, 45a, 46ac, 47a, 48a, 49a, 50a, 51a, 52a, 54a, 56ace, 57ace, 58bd, 59bdf, 60bdf, 61ace, 62ace, 63ac, 64ace, 65aceg, 66bdf, 67ace, 68b.
Macro referenced in scrap 33a.

"shared.index" 57b ≡

```
fROUND dROUND fNINT dNINT
```

◇

File defined by scraps 8c, 11a, 12c, 15a, 18b, 23df, 25c, 29c, 30c, 32d, 34c, 36b, 38b, 39b, 40bd, 41ace, 43b, 45b, 46bd, 47b, 48b, 49b, 50b, 51b, 52b, 54b, 56bdf, 57bd, 58ac, 59ace, 60aceg, 61bdf, 62bdf, 63bd, 64bdf, 65bdf, 66aceg, 67bd, 68ac.

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.

⟨Opcode names 57c⟩ ≡

```
iABS, fABS, dABS, cABS, qABS,
```

◇

Macro defined by scraps 38a, 39a, 40ace, 41bd, 42, 43a, 45a, 46ac, 47a, 48a, 49a, 50a, 51a, 52a, 54a, 56ace, 57ace, 58bd, 59bdf, 60bdf, 61ace, 62ace, 63ac, 64ace, 65aceg, 66bdf, 67ace, 68b.
Macro referenced in scrap 33a.

"shared.index" 57d ≡

```
iABS fABS dABS cABS qABS
```

◇

File defined by scraps 8c, 11a, 12c, 15a, 18b, 23df, 25c, 29c, 30c, 32d, 34c, 36b, 38b, 39b, 40bd, 41ace, 43b, 45b, 46bd, 47b, 48b, 49b, 50b, 51b, 52b, 54b, 56bdf, 57bd, 58ac, 59ace, 60aceg, 61bdf, 62bdf, 63bd, 64bdf, 65bdf, 66aceg, 67bd, 68ac.

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.

⟨Opcode names 57e⟩ ≡

```
iSL, iSLI, iSR, iSRI,
```

◇

Macro defined by scraps 38a, 39a, 40ace, 41bd, 42, 43a, 45a, 46ac, 47a, 48a, 49a, 50a, 51a, 52a, 54a, 56ace, 57ace, 58bd, 59bdf, 60bdf, 61ace, 62ace, 63ac, 64ace, 65aceg, 66bdf, 67ace, 68b.
Macro referenced in scrap 33a.

"shared.index" 58a ≡

```
iSL iSLI iSR iSRI
◇
```

File defined by scraps 8c, 11a, 12c, 15a, 18b, 23df, 25c, 29c, 30c, 32d, 34c, 36b, 38b, 39b, 40bd, 41ace, 43b, 45b, 46bd, 47b, 48b, 49b, 50b, 51b, 52b, 54b, 56bdf, 57bd, 58ac, 59ace, 60aceg, 61bdf, 62bdf, 63bd, 64bdf, 65bdf, 66aceg, 67bd, 68ac.

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.

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

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

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

⟨Opcode names 58b⟩ ≡

```
lSHIFT, lSL, lSLI, lSR, lSRI,
◇
```

Macro defined by scraps 38a, 39a, 40ace, 41bd, 42, 43a, 45a, 46ac, 47a, 48a, 49a, 50a, 51a, 52a, 54a, 56ace, 57ace, 58bd, 59bdf, 60bdf, 61ace, 62ace, 63ac, 64ace, 65aceg, 66bdf, 67ace, 68b.
Macro referenced in scrap 33a.

"shared.index" 58c ≡

```
lSHIFT lSL lSLI lSR lSRI
◇
```

File defined by scraps 8c, 11a, 12c, 15a, 18b, 23df, 25c, 29c, 30c, 32d, 34c, 36b, 38b, 39b, 40bd, 41ace, 43b, 45b, 46bd, 47b, 48b, 49b, 50b, 51b, 52b, 54b, 56bdf, 57bd, 58ac, 59ace, 60aceg, 61bdf, 62bdf, 63bd, 64bdf, 65bdf, 66aceg, 67bd, 68ac.

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*

⟨Opcode names 58d⟩ ≡

```
lAND, lOR, lNAND, lNOR, lEQV, lXOR, lNOT,
◇
```

Macro defined by scraps 38a, 39a, 40ace, 41bd, 42, 43a, 45a, 46ac, 47a, 48a, 49a, 50a, 51a, 52a, 54a, 56ace, 57ace, 58bd, 59bdf, 60bdf, 61ace, 62ace, 63ac, 64ace, 65aceg, 66bdf, 67ace, 68b.
Macro referenced in scrap 33a.

"shared.index" 59a ≡

```
lAND lOR lNAND lNOR lEQV lXOR lNOT
```
◇

File defined by scraps 8c, 11a, 12c, 15a, 18b, 23df, 25c, 29c, 30c, 32d, 34c, 36b, 38b, 39b, 40bd, 41ace, 43b, 45b, 46bd, 47b, 48b, 49b, 50b, 51b, 52b, 54b, 56bdf, 57bd, 58ac, 59ace, 60aceg, 61bdf, 62bdf, 63bd, 64bdf, 65bdf, 66aceg, 67bd, 68ac.

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()`

⟨Opcode names 59b⟩ ≡

```
iMOD, fMOD, dMOD,
```
◇

Macro defined by scraps 38a, 39a, 40ace, 41bd, 42, 43a, 45a, 46ac, 47a, 48a, 49a, 50a, 51a, 52a, 54a, 56ace, 57ace, 58bd, 59bdf, 60bdf, 61ace, 62ace, 63ac, 64ace, 65aceg, 66bdf, 67ace, 68b.
Macro referenced in scrap 33a.

"shared.index" 59c ≡

```
iMOD fMOD dMOD
```
◇

File defined by scraps 8c, 11a, 12c, 15a, 18b, 23df, 25c, 29c, 30c, 32d, 34c, 36b, 38b, 39b, 40bd, 41ace, 43b, 45b, 46bd, 47b, 48b, 49b, 50b, 51b, 52b, 54b, 56bdf, 57bd, 58ac, 59ace, 60aceg, 61bdf, 62bdf, 63bd, 64bdf, 65bdf, 66aceg, 67bd, 68ac.

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()`

⟨Opcode names 59d⟩ ≡

```
iSIGN, fSIGN, dSIGN,
```
◇

Macro defined by scraps 38a, 39a, 40ace, 41bd, 42, 43a, 45a, 46ac, 47a, 48a, 49a, 50a, 51a, 52a, 54a, 56ace, 57ace, 58bd, 59bdf, 60bdf, 61ace, 62ace, 63ac, 64ace, 65aceg, 66bdf, 67ace, 68b.
Macro referenced in scrap 33a.

"shared.index" 59e ≡

```
iSIGN fSIGN dSIGN
```
◇

File defined by scraps 8c, 11a, 12c, 15a, 18b, 23df, 25c, 29c, 30c, 32d, 34c, 36b, 38b, 39b, 40bd, 41ace, 43b, 45b, 46bd, 47b, 48b, 49b, 50b, 51b, 52b, 54b, 56bdf, 57bd, 58ac, 59ace, 60aceg, 61bdf, 62bdf, 63bd, 64bdf, 65bdf, 66aceg, 67bd, 68ac.

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*

⟨Opcode names 59f⟩ ≡

```
iDIM, fDIM, dDIM,
```
◇

Macro defined by scraps 38a, 39a, 40ace, 41bd, 42, 43a, 45a, 46ac, 47a, 48a, 49a, 50a, 51a, 52a, 54a, 56ace, 57ace, 58bd, 59bdf, 60bdf, 61ace, 62ace, 63ac, 64ace, 65aceg, 66bdf, 67ace, 68b.
Macro referenced in scrap 33a.

"shared.index" 60a ≡

```
iDIM fDIM dDIM
```

◇

File defined by scraps 8c, 11a, 12c, 15a, 18b, 23df, 25c, 29c, 30c, 32d, 34c, 36b, 38b, 39b, 40bd, 41ace, 43b, 45b, 46bd, 47b, 48b, 49b, 50b, 51b, 52b, 54b, 56bdf, 57bd, 58ac, 59ace, 60aceg, 61bdf, 62bdf, 63bd, 64bdf, 65bdf, 66aceg, 67bd, 68ac.

Double Precision Product

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

⟨Opcode names 60b⟩ ≡

```
dPROD,
```

◇

Macro defined by scraps 38a, 39a, 40ace, 41bd, 42, 43a, 45a, 46ac, 47a, 48a, 49a, 50a, 51a, 52a, 54a, 56ace, 57ace, 58bd, 59bdf, 60bdf, 61ace, 62ace, 63ac, 64ace, 65aceg, 66bdf, 67ace, 68b.
Macro referenced in scrap 33a.

"shared.index" 60c ≡

```
dPROD
```

◇

File defined by scraps 8c, 11a, 12c, 15a, 18b, 23df, 25c, 29c, 30c, 32d, 34c, 36b, 38b, 39b, 40bd, 41ace, 43b, 45b, 46bd, 47b, 48b, 49b, 50b, 51b, 52b, 54b, 56bdf, 57bd, 58ac, 59ace, 60aceg, 61bdf, 62bdf, 63bd, 64bdf, 65bdf, 66aceg, 67bd, 68ac.

Maximum

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

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

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

⟨Opcode names 60d⟩ ≡

```
iMAX, fMAX, dMAX,
```

◇

Macro defined by scraps 38a, 39a, 40ace, 41bd, 42, 43a, 45a, 46ac, 47a, 48a, 49a, 50a, 51a, 52a, 54a, 56ace, 57ace, 58bd, 59bdf, 60bdf, 61ace, 62ace, 63ac, 64ace, 65aceg, 66bdf, 67ace, 68b.
Macro referenced in scrap 33a.

"shared.index" 60e ≡

```
iMAX fMAX dMAX
```

◇

File defined by scraps 8c, 11a, 12c, 15a, 18b, 23df, 25c, 29c, 30c, 32d, 34c, 36b, 38b, 39b, 40bd, 41ace, 43b, 45b, 46bd, 47b, 48b, 49b, 50b, 51b, 52b, 54b, 56bdf, 57bd, 58ac, 59ace, 60aceg, 61bdf, 62bdf, 63bd, 64bdf, 65bdf, 66aceg, 67bd, 68ac.

Minimum

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

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

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

⟨Opcode names 60f⟩ ≡

```
iMIN, fMIN, dMIN,
```

◇

Macro defined by scraps 38a, 39a, 40ace, 41bd, 42, 43a, 45a, 46ac, 47a, 48a, 49a, 50a, 51a, 52a, 54a, 56ace, 57ace, 58bd, 59bdf, 60bdf, 61ace, 62ace, 63ac, 64ace, 65aceg, 66bdf, 67ace, 68b.
Macro referenced in scrap 33a.

"shared.index" 60g ≡

```
iMIN fMIN dMIN
```

◇

File defined by scraps 8c, 11a, 12c, 15a, 18b, 23df, 25c, 29c, 30c, 32d, 34c, 36b, 38b, 39b, 40bd, 41ace, 43b, 45b, 46bd, 47b, 48b, 49b, 50b, 51b, 52b, 54b, 56bdf, 57bd, 58ac, 59ace, 60aceg, 61bdf, 62bdf, 63bd, 64bdf, 65bdf, 66aceg, 67bd, 68ac.

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`

⟨Opcode names 61a⟩ ≡
```
cIMAG, qIMAG,
```
◇

Macro defined by scraps 38a, 39a, 40ace, 41bd, 42, 43a, 45a, 46ac, 47a, 48a, 49a, 50a, 51a, 52a, 54a, 56ace, 57ace, 58bd, 59bdf, 60bdf, 61ace, 62ace, 63ac, 64ace, 65aceg, 66bdf, 67ace, 68b.
Macro referenced in scrap 33a.

"shared.index" 61b ≡
```
cIMAG qIMAG
```
◇

File defined by scraps 8c, 11a, 12c, 15a, 18b, 23df, 25c, 29c, 30c, 32d, 34c, 36b, 38b, 39b, 40bd, 41ace, 43b, 45b, 46bd, 47b, 48b, 49b, 50b, 51b, 52b, 54b, 56bdf, 57bd, 58ac, 59ace, 60aceg, 61bdf, 62bdf, 63bd, 64bdf, 65bdf, 66aceg, 67bd, 68ac.

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

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

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

⟨Opcode names 61c⟩ ≡
```
cCONJ, qCONJ,
```
◇

Macro defined by scraps 38a, 39a, 40ace, 41bd, 42, 43a, 45a, 46ac, 47a, 48a, 49a, 50a, 51a, 52a, 54a, 56ace, 57ace, 58bd, 59bdf, 60bdf, 61ace, 62ace, 63ac, 64ace, 65aceg, 66bdf, 67ace, 68b.
Macro referenced in scrap 33a.

"shared.index" 61d ≡
```
cCONJ qCONJ
```
◇

File defined by scraps 8c, 11a, 12c, 15a, 18b, 23df, 25c, 29c, 30c, 32d, 34c, 36b, 38b, 39b, 40bd, 41ace, 43b, 45b, 46bd, 47b, 48b, 49b, 50b, 51b, 52b, 54b, 56bdf, 57bd, 58ac, 59ace, 60aceg, 61bdf, 62bdf, 63bd, 64bdf, 65bdf, 66aceg, 67bd, 68ac.

Square Root

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

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

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

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

⟨Opcode names 61e⟩ ≡
```
fSQRT, dSQRT, cSQRT, qSQRT,
```
◇

Macro defined by scraps 38a, 39a, 40ace, 41bd, 42, 43a, 45a, 46ac, 47a, 48a, 49a, 50a, 51a, 52a, 54a, 56ace, 57ace, 58bd, 59bdf, 60bdf, 61ace, 62ace, 63ac, 64ace, 65aceg, 66bdf, 67ace, 68b.
Macro referenced in scrap 33a.

"shared.index" 61f ≡
```
fSQRT dSQRT cSQRT qSQRT
```
◇

File defined by scraps 8c, 11a, 12c, 15a, 18b, 23df, 25c, 29c, 30c, 32d, 34c, 36b, 38b, 39b, 40bd, 41ace, 43b, 45b, 46bd, 47b, 48b, 49b, 50b, 51b, 52b, 54b, 56bdf, 57bd, 58ac, 59ace, 60aceg, 61bdf, 62bdf, 63bd, 64bdf, 65bdf, 66aceg, 67bd, 68ac.

Exponential Computes e^x

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

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

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

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

⟨Opcode names 62a⟩ ≡

```
fEXP, dEXP, cEXP, qEXP,
```

◇

Macro defined by scraps 38a, 39a, 40ace, 41bd, 42, 43a, 45a, 46ac, 47a, 48a, 49a, 50a, 51a, 52a, 54a, 56ace, 57ace, 58bd, 59bdf, 60bdf, 61ace, 62ace, 63ac, 64ace, 65aceg, 66bdf, 67ace, 68b.
Macro referenced in scrap 33a.

"`shared.index`" 62b ≡

```
fEXP dEXP cEXP qEXP
```

◇

File defined by scraps 8c, 11a, 12c, 15a, 18b, 23df, 25c, 29c, 30c, 32d, 34c, 36b, 38b, 39b, 40bd, 41ace, 43b, 45b, 46bd, 47b, 48b, 49b, 50b, 51b, 52b, 54b, 56bdf, 57bd, 58ac, 59ace, 60aceg, 61bdf, 62bdf, 63bd, 64bdf, 65bdf, 66aceg, 67bd, 68ac.

Natural Logarithm

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

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

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

`qLOG` *dcomplex_antilog* `=>` *dcomplex_logarithm*

⟨Opcode names 62c⟩ ≡

```
fLOG, dLOG, cLOG, qLOG,
```

◇

Macro defined by scraps 38a, 39a, 40ace, 41bd, 42, 43a, 45a, 46ac, 47a, 48a, 49a, 50a, 51a, 52a, 54a, 56ace, 57ace, 58bd, 59bdf, 60bdf, 61ace, 62ace, 63ac, 64ace, 65aceg, 66bdf, 67ace, 68b.
Macro referenced in scrap 33a.

"`shared.index`" 62d ≡

```
fLOG dLOG cLOG qLOG
```

◇

File defined by scraps 8c, 11a, 12c, 15a, 18b, 23df, 25c, 29c, 30c, 32d, 34c, 36b, 38b, 39b, 40bd, 41ace, 43b, 45b, 46bd, 47b, 48b, 49b, 50b, 51b, 52b, 54b, 56bdf, 57bd, 58ac, 59ace, 60aceg, 61bdf, 62bdf, 63bd, 64bdf, 65bdf, 66aceg, 67bd, 68ac.

Common Logarithm Computes the logarithm, base 10

`fLOG10` *real_antilog* `=>` *real_logarithm*

`dLOG10` *double_antilog* `=>` *double_logarithm*

⟨Opcode names 62e⟩ ≡

```
fLOG10, dLOG10,
```

◇

Macro defined by scraps 38a, 39a, 40ace, 41bd, 42, 43a, 45a, 46ac, 47a, 48a, 49a, 50a, 51a, 52a, 54a, 56ace, 57ace, 58bd, 59bdf, 60bdf, 61ace, 62ace, 63ac, 64ace, 65aceg, 66bdf, 67ace, 68b.
Macro referenced in scrap 33a.

"`shared.index`" 62f ≡

```
fLOG10 dLOG10
```

◇

File defined by scraps 8c, 11a, 12c, 15a, 18b, 23df, 25c, 29c, 30c, 32d, 34c, 36b, 38b, 39b, 40bd, 41ace, 43b, 45b, 46bd, 47b, 48b, 49b, 50b, 51b, 52b, 54b, 56bdf, 57bd, 58ac, 59ace, 60aceg, 61bdf, 62bdf, 63bd, 64bdf, 65bdf, 66aceg, 67bd, 68ac.

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*

⟨Opcode names 63a⟩ ≡

```
iPOW, fPOW, dPOW, cPOW, qPOW,
```
◇

Macro defined by scraps 38a, 39a, 40ace, 41bd, 42, 43a, 45a, 46ac, 47a, 48a, 49a, 50a, 51a, 52a, 54a, 56ace, 57ace, 58bd, 59bdf, 60bdf, 61ace, 62ace, 63ac, 64ace, 65aceg, 66bdf, 67ace, 68b.
Macro referenced in scrap 33a.

"shared.index" 63b ≡

```
iPOW fPOW dPOW cPOW qPOW
```
◇

File defined by scraps 8c, 11a, 12c, 15a, 18b, 23df, 25c, 29c, 30c, 32d, 34c, 36b, 38b, 39b, 40bd, 41ace, 43b, 45b, 46bd, 47b, 48b, 49b, 50b, 51b, 52b, 54b, 56bdf, 57bd, 58ac, 59ace, 60aceg, 61bdf, 62bdf, 63bd, 64bdf, 65bdf, 66aceg, 67bd, 68ac.

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*

⟨Opcode names 63c⟩ ≡

```
fPOWi, dPOWi, cPOWi, qPOWi,
```
◇

Macro defined by scraps 38a, 39a, 40ace, 41bd, 42, 43a, 45a, 46ac, 47a, 48a, 49a, 50a, 51a, 52a, 54a, 56ace, 57ace, 58bd, 59bdf, 60bdf, 61ace, 62ace, 63ac, 64ace, 65aceg, 66bdf, 67ace, 68b.
Macro referenced in scrap 33a.

"shared.index" 63d ≡

```
fPOWi dPOWi cPOWi qPOWi
```
◇

File defined by scraps 8c, 11a, 12c, 15a, 18b, 23df, 25c, 29c, 30c, 32d, 34c, 36b, 38b, 39b, 40bd, 41ace, 43b, 45b, 46bd, 47b, 48b, 49b, 50b, 51b, 52b, 54b, 56bdf, 57bd, 58ac, 59ace, 60aceg, 61bdf, 62bdf, 63bd, 64bdf, 65bdf, 66aceg, 67bd, 68ac.

Sine

`fSIN` *real_input* `=>` *real_result*

`dSIN` *double_input* `=>` *double_result*

`cSIN` *complex_input* `=>` *complex_result*

`qSIN` *dcomplex_input* `=>` *dcomplex_result*

⟨Opcode names 64a⟩ ≡
```
fSIN, dSIN, cSIN, qSIN,
```
◇

Macro defined by scraps 38a, 39a, 40ace, 41bd, 42, 43a, 45a, 46ac, 47a, 48a, 49a, 50a, 51a, 52a, 54a, 56ace, 57ace, 58bd, 59bdf, 60bdf, 61ace, 62ace, 63ac, 64ace, 65aceg, 66bdf, 67ace, 68b.
Macro referenced in scrap 33a.

"shared.index" 64b ≡
```
fSIN dSIN cSIN qSIN
```
◇

File defined by scraps 8c, 11a, 12c, 15a, 18b, 23df, 25c, 29c, 30c, 32d, 34c, 36b, 38b, 39b, 40bd, 41ace, 43b, 45b, 46bd, 47b, 48b, 49b, 50b, 51b, 52b, 54b, 56bdf, 57bd, 58ac, 59ace, 60aceg, 61bdf, 62bdf, 63bd, 64bdf, 65bdf, 66aceg, 67bd, 68ac.

Cosine

`fCOS` *real_input* `=>` *real_result*

`dCOS` *double_input* `=>` *double_result*

`cCOS` *complex_input* `=>` *complex_result*

`qCOS` *dcomplex_input* `=>` *dcomplex_result*

⟨Opcode names 64c⟩ ≡
```
fCOS, dCOS, cCOS, qCOS,
```
◇

Macro defined by scraps 38a, 39a, 40ace, 41bd, 42, 43a, 45a, 46ac, 47a, 48a, 49a, 50a, 51a, 52a, 54a, 56ace, 57ace, 58bd, 59bdf, 60bdf, 61ace, 62ace, 63ac, 64ace, 65aceg, 66bdf, 67ace, 68b.
Macro referenced in scrap 33a.

"shared.index" 64d ≡
```
fCOS dCOS cCOS qCOS
```
◇

File defined by scraps 8c, 11a, 12c, 15a, 18b, 23df, 25c, 29c, 30c, 32d, 34c, 36b, 38b, 39b, 40bd, 41ace, 43b, 45b, 46bd, 47b, 48b, 49b, 50b, 51b, 52b, 54b, 56bdf, 57bd, 58ac, 59ace, 60aceg, 61bdf, 62bdf, 63bd, 64bdf, 65bdf, 66aceg, 67bd, 68ac.

Tangent

`fTAN` *real_input* `=>` *real_result*

`dTAN` *double_input* `=>` *double_result*

⟨Opcode names 64e⟩ ≡
```
fTAN, dTAN,
```
◇

Macro defined by scraps 38a, 39a, 40ace, 41bd, 42, 43a, 45a, 46ac, 47a, 48a, 49a, 50a, 51a, 52a, 54a, 56ace, 57ace, 58bd, 59bdf, 60bdf, 61ace, 62ace, 63ac, 64ace, 65aceg, 66bdf, 67ace, 68b.
Macro referenced in scrap 33a.

"shared.index" 64f ≡
```
fTAN dTAN
```
◇

File defined by scraps 8c, 11a, 12c, 15a, 18b, 23df, 25c, 29c, 30c, 32d, 34c, 36b, 38b, 39b, 40bd, 41ace, 43b, 45b, 46bd, 47b, 48b, 49b, 50b, 51b, 52b, 54b, 56bdf, 57bd, 58ac, 59ace, 60aceg, 61bdf, 62bdf, 63bd, 64bdf, 65bdf, 66aceg, 67bd, 68ac.

Arcsine

`fASIN` *real_input* `=>` *real_result*

`dASIN` *double_input* `=>` *double_result*

⟨Opcode names 65a⟩ ≡

```
fASIN, dASIN,
```

◇

Macro defined by scraps 38a, 39a, 40ace, 41bd, 42, 43a, 45a, 46ac, 47a, 48a, 49a, 50a, 51a, 52a, 54a, 56ace, 57ace, 58bd, 59bdf, 60bdf, 61ace, 62ace, 63ac, 64ace, 65aceg, 66bdf, 67ace, 68b.
Macro referenced in scrap 33a.

"shared.index" 65b ≡

```
fASIN dASIN
```

◇

File defined by scraps 8c, 11a, 12c, 15a, 18b, 23df, 25c, 29c, 30c, 32d, 34c, 36b, 38b, 39b, 40bd, 41ace, 43b, 45b, 46bd, 47b, 48b, 49b, 50b, 51b, 52b, 54b, 56bdf, 57bd, 58ac, 59ace, 60aceg, 61bdf, 62bdf, 63bd, 64bdf, 65bdf, 66aceg, 67bd, 68ac.

Arccosine

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

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

⟨Opcode names 65c⟩ ≡

```
fACOS, dACOS,
```

◇

Macro defined by scraps 38a, 39a, 40ace, 41bd, 42, 43a, 45a, 46ac, 47a, 48a, 49a, 50a, 51a, 52a, 54a, 56ace, 57ace, 58bd, 59bdf, 60bdf, 61ace, 62ace, 63ac, 64ace, 65aceg, 66bdf, 67ace, 68b.
Macro referenced in scrap 33a.

"shared.index" 65d ≡

```
fACOS dACOS
```

◇

File defined by scraps 8c, 11a, 12c, 15a, 18b, 23df, 25c, 29c, 30c, 32d, 34c, 36b, 38b, 39b, 40bd, 41ace, 43b, 45b, 46bd, 47b, 48b, 49b, 50b, 51b, 52b, 54b, 56bdf, 57bd, 58ac, 59ace, 60aceg, 61bdf, 62bdf, 63bd, 64bdf, 65bdf, 66aceg, 67bd, 68ac.

Arctangent

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

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

⟨Opcode names 65e⟩ ≡

```
fATAN, dATAN,
```

◇

Macro defined by scraps 38a, 39a, 40ace, 41bd, 42, 43a, 45a, 46ac, 47a, 48a, 49a, 50a, 51a, 52a, 54a, 56ace, 57ace, 58bd, 59bdf, 60bdf, 61ace, 62ace, 63ac, 64ace, 65aceg, 66bdf, 67ace, 68b.
Macro referenced in scrap 33a.

"shared.index" 65f ≡

```
fATAN dATAN
```

◇

File defined by scraps 8c, 11a, 12c, 15a, 18b, 23df, 25c, 29c, 30c, 32d, 34c, 36b, 38b, 39b, 40bd, 41ace, 43b, 45b, 46bd, 47b, 48b, 49b, 50b, 51b, 52b, 54b, 56bdf, 57bd, 58ac, 59ace, 60aceg, 61bdf, 62bdf, 63bd, 64bdf, 65bdf, 66aceg, 67bd, 68ac.

`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*

⟨Opcode names 65g⟩ ≡

```
fATAN2, dATAN2,
```

◇

Macro defined by scraps 38a, 39a, 40ace, 41bd, 42, 43a, 45a, 46ac, 47a, 48a, 49a, 50a, 51a, 52a, 54a, 56ace, 57ace, 58bd, 59bdf, 60bdf, 61ace, 62ace, 63ac, 64ace, 65aceg, 66bdf, 67ace, 68b.
Macro referenced in scrap 33a.

"shared.index" 66a ≡
```
fATAN2 dATAN2
```
◇

File defined by scraps 8c, 11a, 12c, 15a, 18b, 23df, 25c, 29c, 30c, 32d, 34c, 36b, 38b, 39b, 40bd, 41ace, 43b, 45b, 46bd, 47b, 48b, 49b, 50b, 51b, 52b, 54b, 56bdf, 57bd, 58ac, 59ace, 60aceg, 61bdf, 62bdf, 63bd, 64bdf, 65bdf, 66aceg, 67bd, 68ac.

Hyperbolic Sine

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

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

⟨Opcode names 66b⟩ ≡
```
fSINH, dSINH,
```
◇

Macro defined by scraps 38a, 39a, 40ace, 41bd, 42, 43a, 45a, 46ac, 47a, 48a, 49a, 50a, 51a, 52a, 54a, 56ace, 57ace, 58bd, 59bdf, 60bdf, 61ace, 62ace, 63ac, 64ace, 65aceg, 66bdf, 67ace, 68b.
Macro referenced in scrap 33a.

"shared.index" 66c ≡
```
fSINH dSINH
```
◇

File defined by scraps 8c, 11a, 12c, 15a, 18b, 23df, 25c, 29c, 30c, 32d, 34c, 36b, 38b, 39b, 40bd, 41ace, 43b, 45b, 46bd, 47b, 48b, 49b, 50b, 51b, 52b, 54b, 56bdf, 57bd, 58ac, 59ace, 60aceg, 61bdf, 62bdf, 63bd, 64bdf, 65bdf, 66aceg, 67bd, 68ac.

Hyperbolic Cosine

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

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

⟨Opcode names 66d⟩ ≡
```
fCOSH, dCOSH,
```
◇

Macro defined by scraps 38a, 39a, 40ace, 41bd, 42, 43a, 45a, 46ac, 47a, 48a, 49a, 50a, 51a, 52a, 54a, 56ace, 57ace, 58bd, 59bdf, 60bdf, 61ace, 62ace, 63ac, 64ace, 65aceg, 66bdf, 67ace, 68b.
Macro referenced in scrap 33a.

"shared.index" 66e ≡
```
fCOSH dCOSH
```
◇

File defined by scraps 8c, 11a, 12c, 15a, 18b, 23df, 25c, 29c, 30c, 32d, 34c, 36b, 38b, 39b, 40bd, 41ace, 43b, 45b, 46bd, 47b, 48b, 49b, 50b, 51b, 52b, 54b, 56bdf, 57bd, 58ac, 59ace, 60aceg, 61bdf, 62bdf, 63bd, 64bdf, 65bdf, 66aceg, 67bd, 68ac.

Hyperbolic Tangent

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

`dTANH` *double_input* `=>` *double_result*

⟨Opcode names 66f⟩ ≡
```
fTANH, dTANH,
```
◇

Macro defined by scraps 38a, 39a, 40ace, 41bd, 42, 43a, 45a, 46ac, 47a, 48a, 49a, 50a, 51a, 52a, 54a, 56ace, 57ace, 58bd, 59bdf, 60bdf, 61ace, 62ace, 63ac, 64ace, 65aceg, 66bdf, 67ace, 68b.
Macro referenced in scrap 33a.

"shared.index" 66g ≡
```
fTANH dTANH
```
◇

File defined by scraps 8c, 11a, 12c, 15a, 18b, 23df, 25c, 29c, 30c, 32d, 34c, 36b, 38b, 39b, 40bd, 41ace, 43b, 45b, 46bd, 47b, 48b, 49b, 50b, 51b, 52b, 54b, 56bdf, 57bd, 58ac, 59ace, 60aceg, 61bdf, 62bdf, 63bd, 64bdf, 65bdf, 66aceg, 67bd, 68ac.

B.9 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.

⟨Opcode names 67a⟩ ≡

```
iASRT, iASRTeq, fASRTeq, dASRTeq, cASRTeq, qASRTeq,
```

◇

Macro defined by scraps 38a, 39a, 40ace, 41bd, 42, 43a, 45a, 46ac, 47a, 48a, 49a, 50a, 51a, 52a, 54a, 56ace, 57ace, 58bd, 59bdf, 60bdf, 61ace, 62ace, 63ac, 64ace, 65aceg, 66bdf, 67ace, 68b.
Macro referenced in scrap 33a.

"shared.index" 67b ≡

```
iASRT iASRTeq fASRTeq dASRTeq cASRTeq qASRTeq
```

◇

File defined by scraps 8c, 11a, 12c, 15a, 18b, 23df, 25c, 29c, 30c, 32d, 34c, 36b, 38b, 39b, 40bd, 41ace, 43b, 45b, 46bd, 47b, 48b, 49b, 50b, 51b, 52b, 54b, 56bdf, 57bd, 58ac, 59ace, 60aceg, 61bdf, 62bdf, 63bd, 64bdf, 65bdf, 66aceg, 67bd, 68ac.

B.9.1 Output commands

⟨Opcode names 67c⟩ ≡

```
iOUT, fOUT, dOUT,
```

◇

Macro defined by scraps 38a, 39a, 40ace, 41bd, 42, 43a, 45a, 46ac, 47a, 48a, 49a, 50a, 51a, 52a, 54a, 56ace, 57ace, 58bd, 59bdf, 60bdf, 61ace, 62ace, 63ac, 64ace, 65aceg, 66bdf, 67ace, 68b.
Macro referenced in scrap 33a.

"shared.index" 67d ≡

```
iOUT fOUT dOUT
```

◇

File defined by scraps 8c, 11a, 12c, 15a, 18b, 23df, 25c, 29c, 30c, 32d, 34c, 36b, 38b, 39b, 40bd, 41ace, 43b, 45b, 46bd, 47b, 48b, 49b, 50b, 51b, 52b, 54b, 56bdf, 57bd, 58ac, 59ace, 60aceg, 61bdf, 62bdf, 63bd, 64bdf, 65bdf, 66aceg, 67bd, 68ac.

B.9.2 Cache commands

⟨Opcode names 67e⟩ ≡

```
FETCHor, FETCHrr, FLUSHor, FLUSHrr, WHEREor, WHERErr,
```

◇

Macro defined by scraps 38a, 39a, 40ace, 41bd, 42, 43a, 45a, 46ac, 47a, 48a, 49a, 50a, 51a, 52a, 54a, 56ace, 57ace, 58bd, 59bdf, 60bdf, 61ace, 62ace, 63ac, 64ace, 65aceg, 66bdf, 67ace, 68b.
Macro referenced in scrap 33a.

"shared.index" 68a ≡

```
FETCH FETCHor FETCHrr FLUSHor FLUSHrr WHEREor WHERErr
```

◇

File defined by scraps 8c, 11a, 12c, 15a, 18b, 23df, 25c, 29c, 30c, 32d, 34c, 36b, 38b, 39b, 40bd, 41ace, 43b, 45b, 46bd, 47b, 48b, 49b, 50b, 51b, 52b, 54b, 56bdf, 57bd, 58ac, 59ace, 60aceg, 61bdf, 62bdf, 63bd, 64bdf, 65bdf, 66aceg, 67bd, 68ac.

B.9.3 Memory copy

This opcode specifies a copy from one memory location to another. This opcode takes five parameters, the source and destination tags, the number of bytes to be copied, the source address, and the destination address.

⟨Opcode names 68b⟩ ≡

```
MCOPY
```

◇

Macro defined by scraps 38a, 39a, 40ace, 41bd, 42, 43a, 45a, 46ac, 47a, 48a, 49a, 50a, 51a, 52a, 54a, 56ace, 57ace, 58bd, 59bdf, 60bdf, 61ace, 62ace, 63ac, 64ace, 65aceg, 66bdf, 67ace, 68b.
Macro referenced in scrap 33a.

"shared.index" 68c ≡

```
MCOPY
```

◇

File defined by scraps 8c, 11a, 12c, 15a, 18b, 23df, 25c, 29c, 30c, 32d, 34c, 36b, 38b, 39b, 40bd, 41ace, 43b, 45b, 46bd, 47b, 48b, 49b, 50b, 51b, 52b, 54b, 56bdf, 57bd, 58ac, 59ace, 60aceg, 61bdf, 62bdf, 63bd, 64bdf, 65bdf, 66aceg, 67bd, 68ac.

B.10 Generating ILOC

This section is intended as a "style guide" for programmers building front-ends that produce ILOC.

Calling conventions are machine dependent. The front end must carefully follow the conventions appropriate to each machine. In particular, parameters passed in registers must be listed in the `JSR` operations and parameters passed on the stack must be explicitly stored.

Use the `LDI` operations when possible (instead of loading from the constant pool) to get best optimization.

Use the `CON` operations when possible to ensure best optimization. Typical uses include loading from the constant pool and loading pointers to parameters from the stack.

When loading and storing from memory, use only the `or` form with a zero offset. More complex forms are machine dependent and may inhibit optimization. They are intended to be generated as a result of optimization.

Don't use `BR`*cc* or `CMP`*cc* operations. Their use is machine dependent and will inhibit optimization. They are intended to be generated as the result of optimization.

Don't use the `iADDI`, `iSUBI`, or `iSLI` operations. Their use is machine-dependent and should be avoided in the front-end. They are intended to be generated as a result of optimization.

B.11 Machine Specifics

Conventions to be followed by *a2i* and the code generators to handle machine-dependent features. Typically, we'll need to specify naming conventions for routine entry points, number of parameters passed in registers and on the stack, and number of special registers (*e.g.*, frame pointer, stack pointer, data area pointer, and so forth).

B.11.1 Simulator

For the *i2c* translator, we need several conventions that will differ from ordinary machines. All arguments will be passed in registers (since Fortran arguments are passed by reference, all the parameters are effectively integers).

In the `FRAME` instruction, the first register is the frame pointer, the second register is the data area pointer. Consider the following example:

```
FRAME 100 => r0 r1 r2 r3 r4
```

In this case, a frame size of 100 bytes is specified, `r0` will be the frame pointer, `r1` will point at the data area, and there are three parameters in the registers `r3`, `r4`, and `r5`.

Naming?

Constant formats (especially floating-point)

B.11.2 IBM RS/6000

Appendix C

Code for the parser

This section contains a whole bunch of random pieces of the parser that we may or may not one day include as part of the actual shared document, but which have to be in there now, as everyone is using these little tidbits.

For now, this section will not have a lot of commentation, because we are still in the process of deciding what goes where...

The plan is to use this section as a testing ground and convenient quick place to put stuff so that no one relying on this header file need wait for us to insert these things in just the right places, with just the right words surrounding them.

⟨Parser-code definitions 70a⟩ ≡

```
/* holds each static code operation, which is any of : xDATA, BYTES, or the
   NAME statement; the next and prev pointers are used to form doubly linked,
   circular lists of operations */
typedef struct static_code_node
{
    Opcode_Names op_name;
    Expr expression;
    Expr repetition_factor; /* set to positive integer for xDATA statements,
                               zero for other static code statements */
    struct static_code_node *next_operation;
    struct static_code_node *prev_operation;

} Static_Code_Node, *Static_Code_Node_Ptr;
◇
```

Macro defined by scraps 70ab, 71abcdefgh, 72abcd, 73ab, 74, 75, 76abc.
Macro referenced in scrap 18c.

⟨Parser-code definitions 70b⟩ ≡

```
/* returns the address of the opcode name (specified by the opcode_names
   enumerated type in opcode.h) from the opcode array (above) */
Opcode *Opcode_Get_Address(Opcode_Names opcode_name);

◇
```

Macro defined by scraps 70ab, 71abcdefgh, 72abcd, 73ab, 74, 75, 76abc.
Macro referenced in scrap 18c.

⟨Parser-code definitions 71a⟩ ≡

```
/* holds a linked list of operations; the next and prev pointers are used to
    form doubly linked, circular lists of operation lists */
typedef struct static_code_list
{
    Static_Code_Node* operation_list;
    Label* label;
    struct static_code_list *next_list;
    struct static_code_list *prev_list;

} Static_Code_List;
◇
```

Macro defined by scraps 70ab, 71abcdefgh, 72abcd, 73ab, 74, 75, 76abc.
Macro referenced in scrap 18c.

⟨Parser-code definitions 71b⟩ ≡

```
extern Static_Code_List *static_code_list;
◇
```

Macro defined by scraps 70ab, 71abcdefgh, 72abcd, 73ab, 74, 75, 76abc.
Macro referenced in scrap 18c.

⟨Parser-code definitions 71c⟩ ≡

```
typedef Unsigned_Int2 Tag;
◇
```

Macro defined by scraps 70ab, 71abcdefgh, 72abcd, 73ab, 74, 75, 76abc.
Macro referenced in scrap 18c.

⟨Parser-code definitions 71d⟩ ≡

```
typedef Unsigned_Int2 Comment_Val;
◇
```

Macro defined by scraps 70ab, 71abcdefgh, 72abcd, 73ab, 74, 75, 76abc.
Macro referenced in scrap 18c.

⟨Parser-code definitions 71e⟩ ≡

```
extern Static_Code_List *NAME_list;
◇
```

Macro defined by scraps 70ab, 71abcdefgh, 72abcd, 73ab, 74, 75, 76abc.
Macro referenced in scrap 18c.

⟨Parser-code definitions 71f⟩ ≡

```
#define FILL_CHAR 0xda
/* this is used by Arena.c; I'm not sure if it matters to anyone else... */
◇
```

Macro defined by scraps 70ab, 71abcdefgh, 72abcd, 73ab, 74, 75, 76abc.
Macro referenced in scrap 18c.

⟨Parser-code definitions 71g⟩ ≡

```
extern Opcode_Names *op_list;
◇
```

Macro defined by scraps 70ab, 71abcdefgh, 72abcd, 73ab, 74, 75, 76abc.
Macro referenced in scrap 18c.

⟨Parser-code definitions 71h⟩ ≡

```
extern Unsigned_Int line_number;
◇
```

Macro defined by scraps 70ab, 71abcdefgh, 72abcd, 73ab, 74, 75, 76abc.
Macro referenced in scrap 18c.

⟨Parser-code definitions 72a⟩ ≡

```
Char *Comment_Get_String(Comment_Val);
```

◇

Macro defined by scraps 70ab, 71abcdefgh, 72abcd, 73ab, 74, 75, 76abc.
Macro referenced in scrap 18c.

⟨Parser-code definitions 72b⟩ ≡

```
Label *Label_Invent(Void);
```

◇

Macro defined by scraps 70ab, 71abcdefgh, 72abcd, 73ab, 74, 75, 76abc.
Macro referenced in scrap 18c.

⟨Parser-code definitions 72c⟩ ≡

```

Void Block_Insert_Instruction(Inst *new_inst, Inst *current_inst);
```

◇

Macro defined by scraps 70ab, 71abcdefgh, 72abcd, 73ab, 74, 75, 76abc.
Macro referenced in scrap 18c.

⟨Parser-code definitions 72d⟩ ≡

```
Block *Block_Build_Drone_Block();
```

◇

Macro defined by scraps 70ab, 71abcdefgh, 72abcd, 73ab, 74, 75, 76abc.
Macro referenced in scrap 18c.

⟨Parser-code definitions 73a⟩ ≡

```
/* Creation date: 9/28/94 TJHARVEY */
/* this module contains the calls necessary to support the linked-list
   data structure used to hole the information from the STACK and GLOBAL
   statements. */

/* this data structure is used to support both GLOBAL and STACK statements;
   since STACK statements don't include the global_label, this value will
   be zero for nodes representing STACK statements */
typedef struct stack_def_node_type Stack_Def_Node;
struct stack_def_node_type{
    Opcode_Names opcode;
    Expr name;
    Expr offset;
    Expr global_label;
    Boolean is_scalar;
    Stack_Def_Node *next;
};

extern Stack_Def_Node *list_of_STACKs;
extern Stack_Def_Node *list_of_GLOBALs;

/* This routine adds a new node onto a list of nodes */
/* for STACK statements, the value sent in for the parameter global_label
   should be zero */
extern Void Stack_Def_Insert(Stack_Def_Node **list, Opcode_Names opcode,
                             Expr offset, Expr name, Expr global_label);

/* This routine prints out all of the nodes in the list of nodes */
extern Void Stack_Def_Put_All(File output_file, Stack_Def_Node *list);

/* This macro simulates a C for-loop and is analogous to the other
   loops such as ForAllBlocks */
/* Usage: Stack_Def_ForAllNodes(Stack_Def_Node *runner, Stack_Def_Node *list)*/
#define Stack_Def_ForAllNodes(runner, list) \
    for (runner=list; runner; runner = runner->next)
```

◇

Macro defined by scraps 70ab, 71abcdefgh, 72abcd, 73ab, 74, 75, 76abc.
Macro referenced in scrap 18c.

To handle C function pointers, we have to have already seen a declaration of the function, or the code produced by `i2c` will be incorrect. We use the **XFUNC** statements to pass along all of the external-function prototypes that `i2c` will have to see in order to produce correct code.

Since `i2c` is the only pass that will look at these, and because they will only be iterated through once, we will store them in a simple linked list.

⟨Parser-code definitions 73b⟩ ≡

```
typedef struct xfunc_node Xfunc_Node;
struct xfunc_node {
    Opcode_Names opcode;
    Expr function_name;
    Xfunc_Node *next;
};
extern Xfunc_Node *list_of_function_names;
```

◇

Macro defined by scraps 70ab, 71abcdefgh, 72abcd, 73ab, 74, 75, 76abc.
Macro referenced in scrap 18c.

⟨Parser-code definitions 74⟩ ≡

```
/* returns a new tag name which is guaranteed to be unique */
Expr Tag_Create(Void);
◇
```

Macro defined by scraps 70ab, 71abcdefgh, 72abcd, 73ab, 74, 75, 76abc.
Macro referenced in scrap 18c.

⟨Parser-code definitions 75⟩ ≡

```
/* this data structure holds the alias names for a tag in a linked list */
typedef struct alias_node Alias_Node;
struct alias_node
{
    Tag alias_name;
    Alias_Node *next;

};



/* this data structure holds the original tag name; it's set up to exist
   in a hash table (thus, the "next" pointer), and to have hanging off of
   it a list of the tag's aliases; the "alias_list_back" pointer is so
   that we can enter the aliases into the list in the same order that
   we read them in, while the "alias_list_front" is so that we can
   output them in the same order that we read them in */
typedef struct tag_node Tag_Node;
struct tag_node
{
    Tag tag_name;
    Unsigned_Int2 alias_count;
    Boolean accessed; /* initialized to FALSE on creation of a new tag_node;
                         gets set to TRUE when the alias list is accessed
                         by an executable operation - note that (currently)
                         this is used only by the parser */
    Boolean added_after_accessed; /* initialized to FALSE on creation of
                                     a new tag node; gets set to TRUE if
                                     an alias is added for this tag after
                                     an executable operation has referenced
                                     it */
    Alias_Node *alias_list_front;
    Alias_Node *alias_list_back;
    Tag_Node *next;
    /* "next" pointer for the hash bucket */
    Tag_Node *canonical_next;
    /* "next" pointer for our global list of values; this linked list
       hooks together all of the tag_nodes so that we can print them
       out easily */

};



/* supercedes the Alias_Store_Tag call; this does all of the allocation
   and entry into the hash table automatically; if the tag already
   has an entry in hash table, the pointer to this data structure is
   returned */
Tag_Node *Alias_Install(Tag tag_name);


/* adds an "alias_node" onto the tag's list */
Void Alias_Add(Tag_Node *tag, Tag alias);
◇
```

Macro defined by scraps 70ab, 71abcdefgh, 72abcd, 73ab, 74, 75, 76abc.
Macro referenced in scrap 18c.

⟨Parser-code definitions 76a⟩ ≡

```
/* this for loop runs through the list of aliases for a tag */
/*
 * Alias_ForAllAliases(Alias_Node *next_alias_node, Tag_Node *tag_node)
 */
#define Alias_ForAllAliases(next_alias, tag_node) \
    for (next_alias = tag_node->alias_list_front; \
         next_alias != NULL;                      \
         next_alias = next_alias->next)
```

◇

Macro defined by scraps 70ab, 71abcdefgh, 72abcd, 73ab, 74, 75, 76abc.
Macro referenced in scrap 18c.

⟨Parser-code definitions 76b⟩ ≡

```
/* returns a pointer to the tag node */
Tag_Node *Alias_Retrieve_Tag(Tag tag_name);
```

◇

Macro defined by scraps 70ab, 71abcdefgh, 72abcd, 73ab, 74, 75, 76abc.
Macro referenced in scrap 18c.

⟨Parser-code definitions 76c⟩ ≡

◇

Macro defined by scraps 70ab, 71abcdefgh, 72abcd, 73ab, 74, 75, 76abc.
Macro referenced in scrap 18c.

Appendix D

Identifier Index