Introduction
Program Representation
- Symbol Tables
- Primitive Types
- Constants
- Identifiers
- Pseudo-registers
- Special Registers
- Statement Labels
- Storage Accesses
- Aggregates

Instruction Specification
- Storage Access Instructions
- Leaf Opcodes
- Unary Expression Opcodes
- Type Conversion Expression Opcodes
- Binary Expression Opcodes
- Ternary Expression Opcodes
- N-ary Expression Opcodes
- Control Flow Statements
    - Hierarchical control flow statements
    - Flat control flow statements
- Call Statements
- Java Call Statements
- Calls with Return Values Assigned
- Exceptions Handling
- Memory Allocation and Deallocation
- Other Statements

Declaration Specification
- Module Declaration
- Variable Declaration
- Pseudo-register Declarations
- Type Specification
- Incomplete Type Specification
- Type Declaration
- Java Class and Interface Declaration
- Function Declaration
- Initializations
- Type Parameters

Examples
- Example 1
- Example 2
- Example 3
- Example 4

Introduction

MAPLE IR is an internal program representation to support program compilation and execution. Because any information in the source program may be useful for program analysis and optimization, MAPLE IR aims to provide information about the source program that is as complete as possible.

Program information consists of two parts: declaration part for defining the program constructs and the executable program code. The former is commonly referred to as the *symbol table*.

MAPLE IR is target-independent. It is not pre-disposed towards any specific target processor or processor characteristic.

MAPLE IR can be regarded as the ISA of a virtual machine (VM). The MAPLE VM can be regarded as a general purpose processor that takes MAPLE IR as input and directly executes the program portion of the MAPLE IR. MAPLE VM can be regarded as the first consumer of MAPLE IR. A program compiled into MAPLE IR can be executed on MAPLE VM without the need to finish its compilation to the machine instructions of any target processor.

MAPLE IR is the common representation for programs compiled from different programming languages, which include general purpose languages like C, C++ and Java. MAPLE IR is extensible. As additional languages, including domain-specific languages, are compiled into MAPLE IR, more constructs will be added to represent constructs unique to each language.

MAPLE IR is capable of supporting all known program analyses and optimizations, owing to its flexibility of being able to represent program code at different semantic levels.   MAPLE IR's program representation at the highest level exhibits the following characteristics:

- Many language constructs
- Short code sequences
- Constructs are hierarchical
- No loss of program information

Language-specific analyses and optimizations are best performed at the high level. As compilation proceeds, MAPLE IR is gradually lowered so that the granularity of its operations corresponds more closely to the instructions of a general purpose processor. At the lower levels, general purpose optimizations are performed. In particular, at the lowest level, MAPLE IR instructions map one-to-one to machine instructions most of the time, for the mainstream processor ISAs. This is important to maximize the effects of optimizations at the IR level, as each eliminated operation will have the corresponding effect on the target machine. At the lowest level, all operations, including type conversion operations, are explicitly expressed at the IR level.

MAPLE IR represents program code intrinsically in the form of trees. At the highest level, it honors the hierarchical form of the program as it exists at the source level via the tree representation. It also honors the abstract operations defined by the language. As compilation proceeds, the abstract operations are lowered into general-purpose operations that require longer code sequences. The program structure also becomes more flat, as general-purpose processors work by executing lists of instructions sequentially.

Since MAPLE IR is one IR that can exist at multiple levels of semantics, the level of a MAPLE IR program is dictated by the constraints that it adheres to. These constraints are of the following two types:

- Opcodes allowed - The higher the level, the more types of opcodes allowed, including opcodes generated only from specific languages. At the lowest level, only opcodes that correspond one-to-one to operations in a general purpose processor are allowed.
- Code structure - The program structure is hierarchical at the higher levels. The hierarchical constructs become less and less as lowering proceeds. At the lowest level, the program structure is flat, consisting

of sequences of primitive instructions consumed by the general purpose processor.

Though MAPLE IR is target-independent at the highest level, the lowering process will make it become target-dependent.

Program Representation

There are additional design criteria at the implementation level. To be friendly to compiler developers, MAPLE IR exists in both binary and ASCII forms. Conversion between the two forms can be viewed as assembly and dis-assembly. Thus, the ASCII form can be viewed as the assembler language of the MAPLE VM instructions. The ASCII form is editable, which implies that it is possible to program in MAPLE IR directly. Thus, the ASCII form of MAPLE IR is modeled after a typical C program, which is made up of:

- declaration statements - these represent the symbol table information
- executable statements - these represent the executable program code

But an ASCII MAPLE IR is *not* intended to obey C programming syntax.

The language front-end compiles a source file into a MAPLE IR file. Thus, each MAPLE IR file corresponds to a CU (compilation unit). A CU is made up of declarations at the global scope. Among the declarations are functions, also known as PUs (program units). Inside PUs are declarations at the local scope followed by the executable code of the function.

There are three kinds of executable nodes in MAPLE IR:

- Leaf nodes - Also called terminal nodes, these nodes denote a value at execution time, which may be a constant or the value of a storage unit.
- Expression nodes - An expression node performs an operation on its operands to compute a result. Its result is a function of the values of its operands and nothing else. Each operand can be either a leaf node or another expression node. Expression nodes are the internal nodes of expression trees. The type field in the expression node gives the type associated with the result of the operation.
- Statement nodes - These represent the flow of control. Execution starts at the entry of the function and continues sequentially statement by statement until a control flow statement is executed. Apart from modifying control flow, statements can also modify data storage in the program. A statement nodes has operands that can be leaf, expression or statement nodes.

In all the executable nodes, the *opcode* field specifies the operation of the node, followed by additional field specification relevant to the opcode. The operands for the node are specified inside *parentheses* separated *commas*. The general form is:

opcode fields (opnd0, opnd1, opnd2)

For example, the C statement "a = b" is specified using the direct assignment opcode **dassign** that assigns the rhs operand *b* to *a* .

```
dassign $a (dread i32 $b)
```

To enable easy visualization in the ASCII IR, whenever an operand is *not* a leaf node, we require the start of a new line indented to the right by two spaces relative to the last line. Thus, the statement "a = b + c" is:

```
dassign $a (
  add i32(dread i32 $b, dread i32 $c))
```

and the statement "a = b + c - d" is:

```
dassign $a (
  sub i32(
   add i32(dread i32 $b, dread i32 $c),
   dread i32 $d))
```

The general rules regarding line breaks are as follows:

- Each expression or statement node must occupy its own line, and each line cannot contain more than one expression or statement node.
- When there is at least one operand that is not a leaf node, then all the operands of the current expression or statement node must be specified in separate new lines, including operands that are leaf nodes.
- Comments can be specified via the character '#', which can be regarded as the end of line character by the IR parser.

For human-edited MAPLE IR files, the line breaks are not enforced for expressions, as they do not affect the correctness of the program, since the end of operand specification is indicated by the closing parenthesis. But there must not be more than one statement node per line, because we do not use the ';'character to delimit statement boundary.

Symbol Tables

Program information that is of declarative nature is best stored in the symbol table portion of the IR. Having the executable instructions refer to the symbol tables reduces the amount of information that needs to be stored in the executable instructions. For each declaration scope, there is a main table called the Symbol Table that manages all the user-defined names in that scope. This implies one global Symbol Table and a local Symbol Table for each function declared in the file. The various types of symbol entries correspond to the various types of declarations supported in typical programming languages:

1. Storage variables
2. Types
3. PUs (program units, functions or their prototypes)

The PU table only exists at the global scope.

In ASCII IR, the IR instructions refer to the various symbols by their names. In the binary representation, only the appropriate scope+index needs to be encoded in the instruction.

Primitive Types

Primitive types can be regarded as pre-defined types supported by the execution engine such that they can be directly operated on. They also play a part in conveying the semantics of operations, as addresses are distinct from unsigned integers. The number in the primitive type name indicates the storage size in bits.

The primitive types are:

- no type - void
- signed integers - i8, i16, i32, i64
- unsigned integers - u8, u16, u32, u64
- booleans- u1
- addresses - ptr, ref, I'a32, a64
- floating point numbers - f32, f64
- complex numbers - c64, c128
- javascript types:
    - dynany
    - dynu32
    - dyni32
    - dynundef
    - dynnull
    - dynhole
    - dynbool
    - dynptr
    - dynf64
    - dynf32
    - dynstr
    - dynobj
- SIMD types - (to be defined)
- unknown

An instruction that produces or operates on values must specify the primitive type in the instruction, as the type is not necessarily implied by the opcode. There is the distinction between *result type* and *operand type*. Result type can be regarded as the type of the value as it resides in a machine register, because arithmetic operations in the mainstream processor architectures are mostly register-based. When an instruction only specifies a single type, the type specified applies to both the operands and the result. In the case of instructions where the operand and result types may differ, the type specified is the result type, and a second field specifies the operand type.

Some opcodes are applicable to non-primitive (or derived) types, as in an aggregate assignment. When the type is derived, the designation *agg* can be used. In such cases, the data size can be looked up from the type of the symbol.

The primitive types *ptr* and *ref* are the target-independent types for addresses. *ref* conveys the additional semantics that the address is a reference to a run-time managed block of memory or object in the heap. Uses of *ptr* or *ref* instead of *a32* or *a64* allow the IR to be independent of the target machine by not manifesting the size of addresses until the later target-dependent compilation phases.

The primitive type *unknown* is used by the language front-end when the type of a field in an object has not been fully resolved because the full definition resides in a different compilation unit.

Constants

Constants in MAPLE IR are always of one of the primitive types.

Integer and address (pointer) types can be specified in decimal or in hexadecimal using the 0x prefix.

Floating point types can be specified in hexadecimal or as floating point literal as in standard C.

Single characters enclosed in single quotes can be used for i8 and u8 constants.

String literals are enclosed in double quotes.

For the complex and SIMD types, the group of values are enclosed in square brackets separated by commas.

Identifiers

In ASCII MAPLE IR, standalone identifier names are regarded as keywords of the IR language. To refer to entries in the symbol tables, identifier names must be prefixed. Identifiers prefixed with '$' are global variables and will be looked up in the global Symbol Table. Identifiers prefixed with '%' are local variables and will be looked up in the local Symbol Table. Identifiers prefixed with '&' are function or method names and will be looked up in the Functions Table. The major purpose of these prefixes is to avoid name clash with the keywords (opcode names, etc.) in the IR.

Pseudo-registers

Pseudo-registers can be regarded as local variables of a primitive type whose addresses are never taken. They can be declared implicitly by their appearances. The primitive type associated with a pseudo-register is sticky. With the exception that integer and address types of the same size can be associated with a pseudo-register.

Because pseudo-registers can only be created to store primitive types, the use of field IDs does not apply to them. Pseudo-registers are referred to by the '%' prefix followed by a number. This distinguishes them from other local variables that are not pseudo-registers, as their names cannot start with a number.

The compiler will promote variables to pseudo-registers. To avoid the loss of high level type information when a variable is promoted to pseudo-registers, reg declarations are used to provide the type information associated with the pseudo-registers.

Special Registers

Special registers are registers with special meaning. They are all specified using %% as prefix. %%SP is the stack pointer and %%FP the frame pointer in referencing the stack frame of the current function. %%GP is the global pointer used for addressing global variables.

Special registers %%retval0, %%retval1, %%retval2, etc. are used for fetching the multiple values returned by a call. They are overwritten by each call, and should only be read *at most once* after each call. They can assume whatever is the type of the return value.

Statement Labels

Label names are prefixed with '@' which serves to identify them. Any statement beginning with a label name defines that label as referring to that text position. Labels are only referred to locally by goto and branch statements.

Storage Accesses

Since MAPLE IR is target-independent, it does not exhibit any pre-disposition as to how storage are allocated for the program variables. It only applies rules defined by the language regarding storage.

In general, there are two distinct kinds of storage accesses: direct and indirect. Direct accesses do not require any run-time computation to determine the exact address location. Indirect accesses require address arithmetic before the location can be determined. Indirect accesses are associated with pointer dereferences and arrays. Direct accesses are associated with scalar variables and fixed fields inside structures.

Direct accesses can be mapped to pseudo-register if the variable or field has no alias. Indirect accesses cannot be mapped to pseudo-registers unless the computed address does not change. But since indirect accesses may represent array and matrix elements, there are many optimizations relevant only to indirect storage accesses.

In MAPLE IR, **dassign** and **dread**are the opcodes for direct assignments and direct references; **iassign** and **iread** are the opcodes for indirect assignments and indirect references.

Aggregates

Aggregates (or composites) are either structures or arrays. They both designate a grouping of storage elements. In structures, the storage elements are designated by field names and can be of different types and sizes. In this document, the structure designation includes classes and objects in object-oriented programming languages. In arrays, the same storage element is repeated a number of times and the elements are accessed via index (or subscript).

Arrays

Array subscripting designate address computation. Since making the subscripts stand out facilitate data dependency analysis and other loop nest optimizations, MAPLE IR represents array subscripting using the special **array** opcode, which returns the address resulting from the subscripting operation. For example, "a[i] = i" is:

```
iassign<*i32>(
  array a32 <* [10] i32> (addrof a32 $a, dread i32 $i),              # <* [10]
i32> indicates pointer to an array of 10 ints
  dread i32 $i)
```

and "x = a[i,j]" is:

```
dassign $x (
  iread i32 <* i32>(
   array a32 <* [10] [10] i32> (addrof a32 a, dread i32 i,dread i32 $j))) # <* [10] [10]
i32 indicates pointer to a 10x10 matrix of ints
```

Structures

Fields in a structure can be accessed directly, but use of **dassign** or **dread** on a structure would refer to the entire structure as an aggregate. Thus, we extend **dassign**, **dread**, **iassign** and **iread** to take an additional parameter called field-ID.
In general, for a top level structure, unique field-IDs can be assigned to all the fields contained inside the structure. Field-ID 0 is assigned to the top level structure. (Field-ID is also 0 if it is not a structure.) As each field is visited, the field-ID is incremented by 1. If a field is a structure, that structure is assigned a unique

field-ID, and then field-ID assignments continue with the fields inside the nested structure. If a field is an array, the array is assigned only one field-ID.

Note that if a structure exists both standalone and nested inside another structure, the same field inside the structure will be assigned different field-IDs because field-ID assignment always starts from the top level structure.

Three kinds of structures are supported: **struct, class** and**interface**.

A **struct** corresponds to the struct type in C, and is specified by the **struct** keyword followed by a list of field declarations enclosed by braces, as in:

```
struct{
        @f1 i32,
        @f2 <structz>}    # structz is the type name of another struct
```

A **class** corresponds to the class type in Java, to provide single inheritances. The syntax is the same as **struct** except for an additional type name specified after the **class** keyword that specifies the class it inherits from. Fields in the parent class are also referred to via field-IDs, as if the first field of the derived class is the parent class. In other words, the parent class is treated like a sub-structure.

```
class <classz>{                      # classz is the parent of this class being defined
        @f1 i32,
        @f2 f32 }
```

Unrelated to storage, structures can contain member function prototypes. The list of member function prototypes must appear after all the fields have been specified. Each member function name starts with &, which indicates that it is a function prototype. The prototype specification follows the same syntax as ordinary function prototypes.

An **interface**corresponds to the interface type in Java, and has the same form as **class**, except that it cannot be instantiated via a var declaration, and fields declared inside it are always statically allocated. More details are provided later in this document.

Instruction Specification

In ASCII MAPLE IR, we use parentheses and braces to distinguish operands from the other fields of an instruction, to facilitate visualization of the nested structure of MAPLE IR. The expression operands of each instruction are always enclosed by parentheses, using commas to separate the operands. Statement operands are enclosed by braces, with each statement starting on a new line. MAPLE IR does not allow the use of semicolon to indicate the end of each statement. Each statement must start on a new line.

Storage Access Instructions

A memory access instruction either loads a memory location to a register for further processing, or store a value from register to memory. For load instructions, the result type given in the instruction is the type of the loaded value residing in register. If the memory location is of size smaller than the register size, the value being loaded must be of integer type, and there will be implicit zero- or sign-extension depending on the signedness of the result type.

dassign

syntax: `dassign <var-name> <field-id> (<rhs-expr>)`

<rhs-expr> is computed to return a value, which is then assigned to variable <var-name>. If <field-id> is not 0, then the variable must be a structure, and the assignment only applies to the specified field. If <rhs-expr> is of type *agg*, then the size of the structure must match. If <rhs-expr> is a primitive integer type, the assigned variable may be smaller, resulting in a truncation. If<field-id> is not specified, it is assumed to be 0.

dread

syntax: `dread <prim-type> <var-name> <field-id>`

Variable <var-name> is read from its storage location. If the variable is a structure, then <prim-type> should specify *agg*. If <field-id> is not 0, then the variable must be a structure, and instead of reading the entire variable, only the specified field is read. If the field itself is also a structure, then <prim-type> should also specify *agg*. If <field-id> is not specified, it is assumed to be 0.

iassign

syntax: `iassign <type> <field-id> (<addr-expr>, <rhs-expr>)`

<addr-expr> is computed to return an address. <type> gives the high level type of <addr-expr> and must be a pointer type. <rhs-expr> is computed to return a value, which is then assigned to the location specified by <addr-expr>. If <field-id> is not 0, then the computed address must correspond to a structure, and the assignment only applies to the specified field. If <rhs-expr> is of type *agg*, then the size of the structure must match. The size of the location affected by the assignment is determined by what <type> points to. If <rhs-expr> is a primitive integer type, the assigned location (according to what <type> points to) may be smaller, resulting in a truncation. If <field-id> is not specified, it is assumed to be 0.

iread

syntax: `iread <prim-type> <type> <field-id> (<addr-expr>)`

The content of the location specified by the address computed from the address expression <addr-expr> is read (dereferenced) as the given primitive type. <type> gives the high level type of <addr-expr> and must be a pointer type. If the content dereferenced is a structure (as given by what <type> points to), then <prim-type> should specify *agg*.   If <field-id> is not 0, then <type> must specify pointer to a structure, and instead of reading the entire structure, only the specified field is read. If the field itself is also a structure, then <prim-type> should also specify *agg*. If <field-id> is not specified, it is assumed to be 0.

iassignoff

syntax: `iassignoff <prim-type> <offset> (<addr-expr>, <rhs-expr>)`

<rhs-expr> is computed to return a scalar value, which is then assigned to the memory location formed by the addition of <offset> in bytes to <addr-expr>. <prim-type> gives the type of the stored-to location, which also specifies the size of the affected memory location.

iassignfpoff

syntax: `iassignfpoff <prim-types> <offset> (<rhs-expr>)`

<rhs-expr> is computed to return a scalar value, which is then assigned to the memory location formed by the addition of <offset> in bytes to %%FP. <prim-type> gives the type of the stored-to location, which also specifies the size of the affected memory location. This is the same as **iassignoff**where its <addr-expr> is %%FP.

ireadoff

syntax: `ireadoff <prim-type> <offset> (<addr-expr>)`

<prim-type> must be of scalar type. <offset> in bytes is added to <addr-expr> to form the address of the memory location to be read as the specified scalar type.

ireadfpoff

syntax: `ireadfpoff <prim-type> <offset>`

<prim-type> must be of scalar type. <offset> in bytes is added to %%FP to form the address of the memory location to be read as the specified scalar type. This is the same as **ireadoff** where its <addr-expr> is %%FP.

regassign

syntax: `regassign <prim-type> <register> (<rhs-expr>)`

<rhs-expr> is computed to return a scalar value, which is then assigned to the pseudo or special register given by <register>. <prim-type> gives the type of the register, which also specifies the size of the value being assigned.

regread

syntax: `regread <prim-type> <register>`

The given pseudo or special register is read in the scalar type specified by <prim-type>.

Leaf Opcodes

dread and **regread** are leaf opcodes for reading the contents of variables. The following are additional leaf opcodes:

addrof

syntax: `addrof <prim-type> <var-name> <field-id>`

The address of the variable <var-name> is returned. <prim-type> must be either *ptr, a32* or *a64*. If <field-id> is not 0, then the variable must be a structure, and the address of the specified field is returned instead.

addroflabel

syntax: `addroflabel <prim-type> <label>`

The text address of the label is returned. <prim-type> must be either a*32* or a*64.*

addroffunc

syntax: `addroffunc <prim-type> <function-name>`

The text address of the function is returned. <prim-type> must be either a*32* or a*64.*

conststr

syntax: `conststr <prim-type> <string literal>`

The address of the string literal is returned. <prim-type> must be either *ptr, a32* or *a64*. The string must be stored in read-only memory.

conststr16

syntax: `conststr16 <prim-type> <string literal>`

The address of the string literal composed of 16-bit wide characters is returned. <prim-type> must be either *ptr, a32* or *a64*. The string must be stored in read-only memory.

constval

syntax: `constval <prim-type> <const-value>`

The specified constant value of the given primitive type is returned. Since floating point values cannot be represented in ASCII without loss of accuracy, they can be specified in hexadecimal form, in which case <prim-type> indicates the floating point type.

sizeoftype

syntax: `sizeoftype <int-prim-type> <type>`

The size in bytes of <type> is returned as an integer constant value. Since type size is in general target-dependent, use of this opcode preserves the target independence of the program code.

Unary Expression Opcodes

abs

syntax: `abs <prim-type> (<opnd0>)`

The absolute value of the operand is returned.

bnot

syntax: `bnot <prim-type> (<opnd0>)`

Each bit in the operand is reversed and the resulting value is returned.

extractbits

syntax: `extractbits <int-type> <boffset> <bsize> (<opnd0>)`

The bitfield starting at bit position <boffset> with <bsize> number of bits is extracted and then sign- or zero-extended to form the primitive integer given by <int-type>. The operand must be of integer type and must be large enough to contain the specified bitfield.

iaddrof

syntax: `iaddrof <prim-type> <type> <field-id>(<addr-expr>)`

<type> gives the high level type of <addr-expr> and must be a pointer type. The address of the pointed-to item is returned. <prim-type> must be either *ptr, a32* or *a64*. If <field-id> is not 0, then <type> must specify a pointer to a structure, and the address of the specified field in the structure is returned instead. This operation is of no utility if <fielid_id> is 0, as it will just return the value of <addr-expr>.

lnot

syntax: `lnot <prim-type> (<opnd0>)`

If the operand is not 0, 0 is returned. If the operand is 0, 1 is returned.

neg

syntax: `neg <prim-type> (<opnd0>)`

The operand value is negated and returned.

recip

syntax: `recip <prim-type> (<opnd0>)`

The reciprocal of the operand is returned. <prim-type> must be a floating-point type.

sext

syntax: `sext <signed-int-type> <bsize> (<opnd0>)`

Sign-extend the integer by treating the integer size as being <bsize> bits. This can be regarded as a special case of **extractbits** where the bitfield is in the lowest bits. The primitive type <signed-int-type> stays the same.

sqrt

syntax: `sqrt <prim-type> (<opnd0>)`

The square root of the operand is returned. <prim-type> must be a floating-point type.

zext

syntax: `zext <unsigned-int-type> <bsize> (<opnd0>)`

Zero-extend the integer by treating the integer size as being <bsize> bits. This can be regarded as a special case of **extractbits** where the bitfield is in the lowest bits. The primitive type <unsigned-int-type> stays the same.

Type Conversion Expression Opcodes

Type conversion opcodes are unary in nature. With the exception of **retype**, they all require specifying both the *from* and *to* types in the instruction. Conversions between integer types of different sizes require the **cvt** opcode. Conversion between signed and unsigned integers of the same size does not require any operation, not even **retype**.

ceil

syntax: `ceil <prim-type> <float-type> (<opnd0>)`

The floating point value is rounded towards positive infinity.

cvt

syntax: `cvt <to-type> <from-type> (<opnd0>)`

Convert the operand's value from <from-type> to <to-type>. If the sizes of the two types are the same, the conversion must involve altering the bits.

floor

syntax: `floor <prim-type> <float-type> (<opnd0>)`

The floating point value is rounded towards negative infinity.

retype

syntax: `retype <prim-type> <type> (<opnd0>)`

<opnd0> is converted to <prim-type> which has derived type <type> without changing any bits. The size of <opnd0> and <prim-type> must be the same. <opnd0> may be of aggregate type.

round

syntax: `round <prim-type> <float-type> (<opnd0>)`

The floating point value is rounded to the nearest integer.

trunc

syntax: trunc <prim-type> <float-type> (<opnd0>)

The floating point value is rounded towards zero.

Binary Expression Opcodes

add

syntax: `add <prim-type> (<opnd0>, <opnd1>)`

Perform the addition of the two operands.

ashr

syntax: `ashr <int-type> (<opnd0>, <opnd1>)`

Return <opnd0> with its bits shifted to the right by <opnd1> bits. The high order bits shifted in are set according to the original sign bit.

band

syntax: `band <int-type> (<opnd0>, <opnd1>)`

Perform the bitwise AND of the two operands.

bior

syntax: `bior <int-type> (<opnd0>, <opnd1>)`

Perform the bitwise inclusive OR of the two operands.

bxor

syntax: `bxor <int-type> (<opnd0>, <opnd1>)`

Perform the bitwise exclusive OR of the two operands.

cand

syntax: `cand <int-type> (<opnd0>, <opnd1>)`

Perform the logical AND of the two operands via short-circuiting. If <opnd0> yields 0, <opnd1> is not to be evaluated. The result is either 0 or 1.

cior

syntax: `cior <int-type> (<opnd0>, <opnd1>)`

Perform the logical inclusive OR of the two operands via short-circuiting. If <opnd0> yields 1, <opnd1> is not to be evaluated. The result is either 0 or 1.

cmp

syntax: `cmp <int-type> <opnd-prim-type> (<opnd0>, <opnd1>)`

Performs a comparison between <opnd0> and <opnd1>. If the two operands are equal, return 0. If <opnd0> is less than <opnd1>, return -1. Otherwise, return +1.

cmpg

syntax: `cmpg <int-type> <opnd-prim-type> (<opnd0>, <opnd1>)`

Same as **cmp**, except 1 is returned if any operand is NaN.

cmpl

syntax: `cmpl <int-type> <opnd-prim-type> (<opnd0>, <opnd1>)`

Same as **cmp**, except -1 is returned if any operand is NaN.

depositbits

syntax: `depositbits <int-type> <boffset> <bsize> (<opnd0>, <opnd1>)`

Creates a new integer value by depositing the value of <opnd1> into the range of bits in <opnd0> that starts at bit position <boffset> and runs for <bsize> bits. <opnd0> must be large enough to contain the specified bitfield. Depending on the size of <opnd1> relative to the bitfield, there may be truncation. The rest of the bits in <opnd0> remains unchanged.

div

syntax: `div <prim-type> (<opnd0>, <opnd1>)`

Perform <opnd0> divided by <opnd1> and return the result.

eq

syntax: `eq <int-type> <opnd-prim-type> (<opnd0>, <opnd1>)`

If the two operands are equal, return 1. Otherwise, return 0.

ge

syntax: `ge <int-type> <opnd-prim-type> (<opnd0>, <opnd1>)`

If <opnd0> is greater than or equal to <opnd1>, return 1. Otherwise, return 0.

gt

syntax: `ge <int-type> <opnd-prim-type> (<opnd0>, <opnd1>)`

If <opnd0> is greater than <opnd1>, return 1. Otherwise, return 0.

land

syntax: `land <int-type> (<opnd0>, <opnd1>)`

Perform the logical AND of the two operands. The result is either 0 or 1.

lior

syntax: `lior <int-type> (<opnd0>, <opnd1>)`

Perform the logical inclusive OR of the two operands. The result is either 0 or 1.

le

syntax: `le <int-type> <opnd-prim-type> (<opnd0>, <opnd1>)`

If <opnd0> is less than or equal to <opnd1>, return 1. Otherwise, return 0.

lshr

syntax: `lshr <int-type> (<opnd0>, <opnd1>)`

Return <opnd0> with its bits shifted to the right by <opnd1> bits. The high order bits shifted in are set to 0.

lt

syntax:`lt <int-type> <opnd-prim-type> (<opnd0>, <opnd1>)`

If <opnd0> is less than <opnd1>, return 1. Otherwise, return 0.

max

syntax: `max <prim-type> (<opnd0>, <opnd1>)`

Return the maximum of <opnd0> and <opnd1>.

min

syntax: `min <prim-type> (<opnd0>, <opnd1>)`

Return the minimum of <opnd0> and <opnd1>.

mul

syntax: `mul <prim-type> (<opnd0>, <opnd1>)`

Perform the multiplication of the two operands.

ne

syntax: `ne <int-type> <opnd-prim-type> (<opnd0>, <opnd1>)`

If the two operands are not equal, return 1. Otherwise, return 0.

rem

syntax: `rem <prim-type> (<opnd0>, <opnd1>)`

Return the remainder when <opnd0> is divided by <opnd1>.

shl

syntax: `shl <int-type> (<opnd0>, <opnd1>)`

Return <opnd0> with its bits shifted to the left by <opnd1> bits. The low order bits shifted in are set to 0.

sub

syntax: `sub <prim-type> (<opnd0>, <opnd1>)`

Subtract <opnd1> from <opnd0> and return the result.

Ternary Expression Opcodes

select

syntax: `select <prim-type> (<opnd0>, <opnd1>, <opnd2>)`

<opnd0> must be of integer type. <opnd1> and <opnd2> must be of the type given by <prim-type>. If <opnd0> is not 0, return <opnd1>. Otherwise, return <opnd2>.

N-ary Expression Opcodes

array

syntax: `array <check-flag> <addr-type> <array-type> (<opnd0>, <opnd1>, . . . , <opndn>)`

<opnd0> is the base address of an array in memory. <check-flag> is either 0 or 1, indicating bounds-checking needs not be performed or needs to be performed respectively. <array-type> gives the high-level type of a pointer to the array. Return the address resulting from row-major order multi-dimentional indexing operation, with the indices represented by <opnd1> onwards.

intrinsicop

syntax: `intrinsicop <prim-type> <intrinsic> (<opnd0>, ..., <opndn>)`

<intrinsic> indicates an intrinsic function that has no side effect whose return value depends only on the arguments (a pure function), and thus can be represented as an expression opcode.

intrinsicopwithtype

syntax: `intrinsicopwithtype <prim-type> <type> <intrinsic> (<opnd0>, ..., <opndn>)`

This is the same as **intrinsicop** except that it takes on an additional high level type argument specified by <type>.

Control Flow Statements

Program control flows can be represented by either hierarchical statement structures or a flat list of statements. Hierarchical statement structures are mostly derived from constructs in the source language. Flat control flow statements correspond more closely to processor instructions. Thus, hierarchical statements exist only in high level MAPLE IR. They are lowered to the flat control flow statements in the course of compilation.

A statement block is indicated by multiple statements enclosed inside the braces '{' and '}'. Such statement blocks can appear any where that a statement is allowed. In hierarchical statements, nested statements are specified by such statement blocks.

In MAPLE IR, each statement must start on a new line. The use of semicolon is not needed, or even allowed, to indicate the end of statements.

Hierarchical control flow statements

doloop

syntax:

```
doloop <do-var> (<start-expr>, <cont-expr>, <incr-amt>) {
                    <body-stmts> }
```

<do-var> specifies a local integer scalar variable with no alias. <incr-amt> must be an integer expression. <do-var> is initialized with <start-expr>. <cont-expr> must be a single comparison operation representing the termination test.   The loop body is represented by <body-stmts>, which specifies the list of statements to be executed as long as <cont-expr> evaluates to true. After each execution of the loop body, <do-var> is incremented by <incr-amt> and the loop is tested again for termination.

dowhile

syntax:

```
dowhile {

                <body-stmts>} (

                <cond-expr>)
```

Execute the loop body represented by <body-stmts>, and while <cond-expr> evaluates to non-zero, continue to execute <body-stmts>. Since <cond-expr> is tested at the end of the loop body, the loop body is executed at least once.

foreachelem

syntax:

```
foreach <elem-var> <collection-var> {
                    <body-stmts> }
```

This is an abstract loop form where <collection-var> is an array-like variable representing a collection of uniform elements, and <elem-var> specifies a variable whose type is the element type of <collection-var>. The loop body is represented by <body-stmts>, which specifies the list of statements to be repeated for each element of <collection-var> expressed via <elem-var>. This statement will be lowered to a more concrete loop form based on the type of <collection-var>.

if

syntax:

```
if (<cond-expr>) {

               <then-part> }

            else {

               <else-part>}
```

If <cond-expr> evaluates to non-zero, control flow passes to the <then-part> statements. Otherwise, control flow passes to the <else-part> statements. If there is no else part, "else { <else-part> }" can be omitted.

while

syntax:

```
while (<cond-expr>) {

                <body-stmts>}
```

This implements the while loop. While <cond-expr> evaluates to non-zero, the list of statements represented by <body-stmts> are repeatedly executed. Since <cond-expr> is tested before the first execution, the loop body may execute zero times.

Flat control flow statements

brfalse

syntax: `brfalse <label> (<opnd0>)`

If <opnd0> evaluates to 0, branch to <label>. Otherwise, fall through.

brtrue

syntax:`brtrue <label> (<opnd0>)`

If <opnd0> evaluates to non-0, branch to <label>. Otherwise, fall through.

goto

syntax: `goto <label>`

Transfer control unconditionally to <label>.

multiway

syntax:

```
multiway (<opnd0>) <default-label> {

                (<expr0>): goto <label0>

                (<expr1>): goto <label1>

                        . . .

                (<exprn>): goto <labeln> }
```

<opnd0> must be of type convertible to an integer or a string. Following <default-label> is a table of pairs of expression tags and labels. When executed, it evaluates <opnd0> and then searches the table for a match of the evaluated value of <opnd0> with the evaluated value of each expression tag <expri> in the listed order. On a match, control is transferred to the corresponding label. If no match is found, control is transferred to <default-label>. The evaluation of the expression tags must not incur side effect. Depending on the resolved type of <opnd0>, this statement will be lowered to either the **switch** statement or a cascade of **if** statements.

return

syntax: `return (<opnd0>, . . ., <opndn>)`

Return from the current PU with the multiple return values given by the operands. The list of operands can be empty, which corresponds to no return value. The types of <opndi> must be compatible with the list of return types according to the declaration or prototype of the PU.

switch

syntax:

```
switch (<opnd0>) <default-label> {

                <intconst0>: goto <label0>

                <intconst1>: goto <label1>

                        . . .

                <intconstn>: goto <labeln> }
```

<opnd0> must be of integer type. After <default-label>, it specifies a table of pairs of constant integer values (tags) and labels. When executed, it searches the table for a match with the evaluated value of <opnd0> and transfers control to the corresponding label. If no match is found, control is transferred to <default-label>. There must not be duplicate entries for the constant integer values. It is up to the compiler backend to decide how to actually generate code for this statement after analyzing the tag distribution in the table.

rangegoto

syntax:

```
rangegoto (<opnd0> <tag-offset> {

                <intconst0>: goto <label0>

                <intconst1>: goto <label1>

                        . . .

                <intconstn>: goto <labeln> }
```

This is the lowered form of **switch** that explicitly designates its execution to be handled by the jump table mechanism. <opnd0> must be of integer type. After <tag-offset> follows a table of pairs of constant integer values and labels. In searching the table for a match during execution, the evaluated value of <opnd0> minus <tag-offset> is used during execution so as to cause transfer of control to the corresponding label.   There must be no gap in the constant integer values specified, and a match is guaranteed within the range of specified constant integer values, which means the code generator can omit generation of out-of-range checks. There must be no duplicated entries for the constant integer values.

indexgoto

syntax: `indexgoto (<opnd0> <varname1>`

This is only generated by the compiler as a result of lowering the **switch** statement. <varname> is the name of a compiler-generated symbol designating a static array, or jump table, which is statically initialized to store labels. Each stored label marks the code corresponding to a switch case. Execution of this instruction uses the evaluated value of <opnd0> to index into this array and then transfers control to the label stored at that array element. If the evaluated value of <opnd0> is less than 0 or exceeds the number of entries in the jump table, the behavior is undefined.

Call Statements

There are various flavors of procedure invocation. They only specify the actual parameters are passed. Any return value needs to be separately fetched via the special registers %%retval0, %%retval1, %%retval2, etc.

call

syntax: `call <PU-name> (<opnd0>, ..., <opndn>)`

Invoke the PU while passing the parameters given by the operands.

callinstant

syntax: `callinstant <generic-PU-name> <instant-vector> (<opnd0>, ..., <opndn>)`

Instantiate the given generic function according to instantiation vector <instant-vector> and then invoke the PU while passing the parameters given by the operands. See section *Initializations* about instantiation of generic functions.

icall

syntax: `icall (<PU-ptr>, <opnd0>, ..., <opndn>)`

Invoke the PU specified indirectly by <PU-ptr>, passing the parameters given by <opnd0> onwards.

intrinsiccall

syntax: `intrinsiccall <intrinsic> (<opnd0>, ..., <opndn>)`

Invoke the specified intrinsic defined by the compiler while passing the parameters given by the operands.

xintrinsiccall

syntax: `xintrinsiccall <user-intrinsic-index> (<opnd0>, ..., <opndn>)`

Invoke the intrinsic specified as an index into a user-defined intrinsic table while passing the parameters given by the operands.

Java Call Statements

The following statements are used to represent Java member function calls that are not yet resolved.

virtualcall

syntax: `virtualcall <method-name> (<object-ptr>, <opnd0>, ..., <opndn>)`

<object-ptr> is a pointer to an instance of a class. The class hierarchy is searched using the specified <method-name> to find the appropriate virtual method to invoke. The invocation will pass the remaining operands as parameters.

superclasscall

syntax: `superclasscall <method-name> (<object-ptr>, <opnd0>, ..., <opndn>)`

This is the same as **virtualcall** except it will not use the class's own virtual method, but the one in its closest superclass that defines the virtual method.

interfacecall

syntax: `interfacecall <method-name> (<object-ptr>, <opnd0>, ..., <opndn>)`

<method-name> is a method defined in an interface. <object-ptr> is a pointer to an instance of a class the implements the interface. The class is searched using the <method-name> to find the corresponding method to invoke. The invocation will pass the remaining operands as parameters.

There are also **virtualcallinstant**, **superclasscallinstant** and **interfacecallinstant** for calling generic versions of the methods after instantiating with the specified instantiation vector. The instantiation vector is specified after <method-name>, as in the **callinstant**instruction.

Calls with Return Values Assigned

All the various call operations have a corresponding higher-level abstracted variant such that a single call operation also specify how the multiple function return values are assigned, without relying on separate statements to read the %%retval registers. Only assignments to scalar variables or fields in aggregates are allowed. These operations have the same names with the suffix "assigned" added. They are **callassigned, callinstantassigned**, **icallassigned, intrinsiccallassigned, intrinsiccallwithtypeassigned, xintrinsiccallassigned, virtualcallassigned, virtualcallinstantassigned, superclasscallassigned, superclasscallinstantassigned, interfacecallassigned** and **interfacecallinstantassigned**.
Only **callassigned** is defined here. The same extension applies to the definitions of the rest of these call operations.

callassigned

syntax:

```
callassigned <PU-name> (<opnd0>, ..., <opndn>) {

                dassign <var-name0> <field-id0>

                dassign <var-name1> <field-id1>

                      . . .

                dassign <var-namen> <field-idn> }
```

Invoke the PU passing the parameters given by the operands. After returning from the call, the multiple return values are assigned to the scalar variables listed in order. If a specified field-id is not 0, then the corresponding variable must be an aggregate, and the assignment is to the field corresponding to the field-id. If a field-id is absent, 0 is implied. If <var-name> is absent, it means the corresponding return value is ignored by the caller. If a call has no return value, no dassign should be listed.

In the course of compilation, these call instructions may be lowered to use the special registers %%retval0, %%retval1, %%retval2, etc. to indicate how their return values are fetched and used. These special registers are overwritten by each call. The same special register can assume whatever is the type of the return value. Each special register can be read only once after each call.

Exceptions Handling

Described in this section are the various exception handling constructs and operations. The **try** statement marks the entrance to a try block. The **catch** statement marks the entrance to a catch block. The **finally** statement marks the entrance to a finally block. The **endtry** statement marks the end of the composite exception handling constructs that began with the **try**. In addition, there are two special types of labels. Handler labels are placed before **catch** statements, and finally labels are placed before **finally** statements. Handler labels are distinguished from ordinary labels via the prefix "@h@", while finally labels use the prefix "@f@". These special labels explicitly shows the correspondence of try, catch and finally to each other in each try-catch-finally composite, without relying on block nesting. The special register %%thrownval contains the value being thrown, which is the operand of the throw operation that raised the current exception.

try

syntax: `try <handler-label> <finally-label>`

Executing this statement indicates entry into a try block. <handler-label> is 0 when there is no catch block associated with the **try**. <finally-label> is 0 when there is no finally block associated with the **try**. Any exception thrown inside this try block will transfer control to these labels, unless another nested try block is entered. A finally block if present must be executed to conclude the execution of the **try** composite constructs regardless of whether exception is thrown or not.

There are three possible scenarios based on the way the try-catch-finally composite is written:

1. try-catch
2. try-finally
3. try-catch-finally

For case 1, if an exception is thrown inside the try block, control is transferred to the handler label that marks the catch statement and the exception is regarded as having been handled. Program flow eventually exits the try block with a **goto** statement to the label that marks the **endtry** statement. If no exception is thrown, the try block is eventually exited via a **goto** statement to the label that marks the **endtry** statement.

For case 2, if an exception is thrown inside the try block, control is transferred to the finally label that marks the finally statement. But the exception is regarded as not having been handled yet, and the search for the throw's upper level handler starts. If no exception is thrown in try block, program flow eventually exits the try block with a **gosub** statement to the finally block. Execution in the finally block ends with a **retsub**, which returns to the try block and the try block is then exited via a**goto** statement to the label that marks the **endtry** statement.

For case 3, if an exception is thrown inside the try block, control is transferred to the handler label that marks the catch statement as in case 1. Execution in the catch block ends with a **gosub** statement to the finally block. Execution in the finally block ends with a **retsub**, which returns to the catch block and the catch block is then exited via a **goto** statement to the label that marks the **endtry** statement.   If no exception is thrown in the try block, program flow eventually exits the try block with a **gosub** statement to the finally block, and execution will continue in the finally block until it executes a **retsub**, at which time it returns to the try block and the try block is then exited via a **goto** statement to the label that marks the **endtry** statement.

throw

syntax: `throw (<opnd0>)`

Raise a user-defined exception with the given exception value. If this statement is nested within a try block, control is then transferred to the label associated with the try, which is either a **catch** statement or a **finally** statement. If this throw statement is nested within a catch block, control is first transferred to the **finally** associated with the catch if any, in which case the finally block will be executed. After it finishes executing the finally block, the search for the throw's upper level handler starts. If this **throw** statement is nested within a finally block, the search for the throw's upper level handler starts right away. If the throw is not nested inside any try block within a function, the system will look for the first enclosing try block by unwinding the call stack. If no try block is found, the program will terminate. Inside the catch block, the thrown exception value can be accessed using the special register %%thrownval.

catch

syntax: `<handler-label> catch`

This marks the start of the catch block associated with a try. The try block associated with this catch block is regarded to have been exited and the exception is regarded as being handled. If no exception is thrown inside the catch block, exit from the catch block is effected by either a **gosub** statement to a finally label, if there is a finally block, or a **goto** statement to **endtry**.

finally

syntax: `<finally-label> finally`

This marks the start of the finally block. The finally block can be entered either via the execution of a **gosub** statement, or a**throw** statement the finally's corresponding try block that has no catch block, or a **throw** statement in the finally's corresponding catch block. The exit from the finally block can be effected by the execution of either a **retsub** or a **throw** statement in the finally block. If the exit is via a **retsub** and if there is outstanding throw yet to be handled, the search for the throw's upper level handler continues.

cleanuptry

syntax: `cleanuptry`
This statement is generated in situations where the control is going to leave the try-catch-finally composite prematurely via jumps unrelated to exception handling. This statement effects the cleanup work related to exception handling for the current try-catch-finally composite.

endtry

syntax: `<label> endtry`
This marks either the end of each try-catch-finally composite or the end of each javatry block.

gosub

syntax: `gosub <finally-label>`

Transfer control to the finally block marked by <finally-label>. This also has the effect of exiting the try block or catch block which this statement belongs. It is like a **goto**, except that the next instruction is saved. Execution will transfer back to the next instruction when a **retsub** statement is executed. This can also be thought of as a **call**, except it uses label name instead of function name, and no passing of parameter or return value is implied.

retsub

syntax: `retsub`

This must only occur as the last instruction inside a finally block. If there is no outstanding throw, control is transferred back to the instruction following the last **gosub**executed. Otherwise the search for the upper level exception handler continues.

Memory Allocation and Deallocation

The following instructions are related to the allocation and de-allocation of dynamic memory during program execution. The instructions with "gc" as prefix are associated with languages with managed runtime environments.

alloca

syntax: `alloca <prim-type> (<opnd0>)`

This returns a pointer to the block of uninitialized memory allocated by adjusting the function stack pointer %%SP, with size in bytes given by <opnd0>. This instruction must only appear as the right hand side of an assignment operation.

decref

syntax: `decref (<opnd0>)`

<opnd0> must be a **dread** or **iread** of a pointer that refers to an object allocated in the run-time-managed part of the heap. It decrements the reference count of the pointed-to object by 1. <opnd0> must be of primitive type *ref*.

decrefwithcheck

syntax: `decrefwithcheck (<opnd0>)`

<opnd0> must be a **dread** or **iread** of a pointer that refers to an object allocated in the run-time-managed part of the heap. It checks if <opnd0>'s value is null. If not null, it is assumed to be a valid pointer and decrements the reference count of the pointed-to object by 1. <opnd0> must be of primitive type *ref*.

free

syntax: `free (<opnd0>)`

The block of memory pointed to by <opnd0> is released so it can be re-allocated by the system for other uses.

gcmalloc

syntax: `gcmalloc <pointer prim-type> <type>`

This requests the memory manager to allocate an object of type <type> with associated meta-data according to the requirements of the managed runtime. The size of the object must be fixed at compilation time. As this returns the pointer to the allocated block, this instruction must only appear as the right hand side of an assignment operation. The managed runtime is responsible for its eventual deallocation.

gcmallocjarray

syntax: `gcmallocjarray <pointer prim-type> <java-array-type> (<opnd0>)`

This requests the memory manager to allocate a java array object as given by <java-array-type>. The allocated storage must be large enough to store the number of array elements as given by <opnd0>. As this returns the pointer to the allocated block, this instruction must only appear as the right hand side of an assignment operation. The managed runtime is responsible for its eventual deallocation, and the block size must remain fixed during its life time.

incref

syntax: `incref (<opnd0>)`

<opnd0> must be a **dread** or **iread** of a pointer that refers to an object allocated in the run-time managed part of the heap. It increments the reference count of the pointed-to object's by 1. <opnd0> must be of primitive type *ref*.

malloc

syntax: `malloc <pointer prim-type> (<opnd0>)`

This requests the system to allocate a block of uninitialized memory with size in bytes given by <opnd0>. As this returns the pointer to the allocated block, this instruction must only appear as the right hand side of an assignment operation. The block of memory remains unavailable for re-use until it is explicitly freed via the **free** instruction.

Other Statements

assertge

syntax: `assertge (<opnd0>, <opnd1>)`

Raise an exception if <opnd0> is not greater than or equal to <opnd1>. This is used for checking if an array index is within range during execution. <opnd0> and <opnd1> must be of the same type.

assertlt

syntax: `assertlt (<opnd0>, <opnd1>)`

Raise an exception if <opnd0> is not less than <opnd1>. This is used for checking if an array index is within range during execution. <opnd0> and <opnd1> must be of the same type.

assertnonnull

syntax: `assertnonnull (<opnd0>)`

Raise an exception if <opnd0> is a null pointer, corresponding to the value 0.

eval

syntax: `eval (<opnd0>)`

<opnd0> is evaluated but the result is thrown away. If <opnd0> contains volatile references, this statement cannot be optimized away.

membaracquire

syntax: `membaracquire`

This instruction acts as both a Load-to-Load barrier and a Load-to-Store barrier: the order between any load instruction before it and any load instruction after it must be strictly followed, and the order between any load instruction before it and any store instruction after it must be strictly followed.

membarrelease

syntax: `membarrelease`

This instruction acts as both a Load-to-Store barrier and a Store-to-Store barrier: the order between any load instruction before it and any store instruction after it must be strictly followed, and the order between any store instruction before it and any store instruction after it must be strictly followed.

membarfull

syntax: `membarfull`

This instruction acts as a barrier to any load or store instruction before it and any load or store instruction after it.

syncenter

syntax: `syncenter (<opnd0>)`

This instruction indicates entry to a region where the object pointed to by the pointer <opnd0> needs to be synchronized for Java multi-threading. This means at any time, there cannot be more than one thread executing in a synchronized region of the same object. Any other thread attempting to enter a synchronized region of the same object will be blocked. For the compiler, it implies a barrier to the backward movement (against the flow of control) of any operation that accesses the object.

syncexit

syntax: `syncexit (<opnd0>)`

This instruction indicates exit from a region where the object pointed to by the pointer <opnd0> needs to be synchronized for Java multi-threading. For the compiler, it implies a barrier to the forward movement (along the flow of control) of any operation that accesses the object.

Declaration Specification

Module Declaration

Each Maple IR file represents a program module, also called compilation unit, that consists of various declarations at the global scope. The following directives appear at the start of the Maple IR file and provide information about the module:

entryfunc

syntax: `entryfunc <func-name>`

This gives the name of the function defined in the module that will serve as the single entry point for the module.

flavor

syntax: `flavor <IR-flavor>`

The IR flavor gives information as to how the IR was produced, which in turn indicates the state of the compilation process.

globalmemmap

syntax: `globalmemmap = [ <initialization-values> ]`

This specifies the static initialization values of the global memory block as a list of space-separated 32-bit integer constants. The amount of initializations should correspond to the memory size given by **globalmemsize**.

globalmemsize

syntax: `globalmemsize <size-in-bytes>`

This gives the size of the global memory block for storing all global static variables.

globalwordstypetagged

syntax: `globalwordstypetagged = [ <word-values> ]`

This specifies a bitvector initialized to the value specified by the list of space-separated 32-bit integer constants. The Nth bit in this bitvector is set to 1 if the Nth word in **globalmemmap** has a type tag, in which case the type tag is at the (N+1)th word.

globalwordsrefcounted

syntax: `globalwordsrefcounted = [ <word-values> ]`

This specifies a bitvector initialized to the value specified by the list of space-separated 32-bit integer constants. The Nth bit in this bitvector is set to 1 if the Nth word in **globalmemmap** is a pointer to a reference-counted dynamically allocated memory block.

id

syntax: `id <id-number>`

This gives the unique module id assigned to the module. This id enables the Maple virtual machine to handle the execution of program code originating from multiple Maple IR modules.

import

syntax: `import "<filename>"`

<filename> is the path name for a MAPLE type file, with suffix .mplt. The contents of this file are imported. Only type declarations are allowed in the .mplt file. This allows the same type declarations to be shared across multiple files, and large volumes of type declarations to be organized by files. Only one level of import is allowed, as .mplt files are not allowed to have import statements. This is used only after all types and symbol names have been fully resolved.

importpath

syntax: `importpath "<path-name>"`

This specifies a directory path for the compiler to look for the imported MAPLE IR files required to complete the compilation. This is used only in the early compilation phases, before all types and symbol names have been fully resolved. Each appearance only specifies one specific path.

numfuncs

syntax: `numfuncs <integer>`

This gives the number of function definitions in the module, excluding function prototypes.

srclang

syntax: `srclang <language>`

This gives the source language that produces the Maple IR module.

Variable Declaration

syntax: `var <id-name> <storage-class> <type> <type-attributes>`

The keyword 'var' designates a declaration statement for a variable. <id-name> specifies its name, prefixed by '$' or '%' based on whether its scope is global or local respectively. <storage-class> is optional and can be extern, fstatic or pstatic. <type> is the type specification. <type-attributes> is optional and specifies additional attributes like volatile, const, alignment, etc. Examples:

var $x extern f32 volatile static

Pseudo-register Declarations

syntax: `reg <preg-name> <type>`

The keyword 'reg' designates a declaration statement for a pseudo-register. <preg-name> specifies the pseudo-register prefixed by '%'. <type> is the high level type information. If a pseudo-register is only of primitive type, its declaration is optional.

Type Specification

Types are either primitive or derived. Derived types are specified using C-like tokens, except that the specification is always right-associative, following a strictly left to right order. Derived types are distinguished from primitive types by being enclosed in angular brackets. Derived types can also be thought of as high-level types. Examples:

```
var %p <* i32>              # pointer to a 32-bit integer
var %a <[10] i32>            # an array of 10 32-bit integers
var %foo <func(i32) i32>     # a pointer to a function that takes one i32 parameter and
returns an i32 value (func is a keyword)
```

Additional nested angular brackets are not required, as there is no ambiguity due to the right-associative rule. But the nested angular brackets can be optionally inserted to aid readability. Thus, the following two examples are equivalent:

```
var %q <* <[10] i32>>        # pointer to an array of 10 32-bit integers
var %q <* [10] i32>          # pointer to an array of 10 32-bit integers
```

Inside a struct declaration, field names are prefixed by @. Though label names also use @ as prefix, there is no ambiguity due to the usage context between struct field declaration and label usage being distinct. Example:

```
var %s <struct{@f1 i32,
         @f2 f64,
         @f3:3 i32}>          # a bitfield of 3 bits
```

A union declaration has the same syntax as struct. In a union, all the fields overlap with each other.

The last field of a struct can be a flexible array member along the line of the C99 standard, which is an array with variable number of elements. It is specified with empty square brackets, as in:

```
var %p <* struct{@f1 i32,
         @f2 <[] u16>}>          a flexible array member with unsigned 16-bit integers as
elements
```

Structs with flexible array member as its last field can only be dynamically allocated. Its actual size is fixed only at its allocation during execution time, and cannot change during its life time. A struct with flexible array member cannot be nested inside another aggregate. During compilation, the flexible array member is regarded as having size zero. Because its use is usually associated with managed runtime, a language processor may introduce additional meta-data associated with the array. In particular, there must be some language-dependent scheme to keep track of the size of the array during execution time.

When a type needs to be qualified by additional attributes for const, volatile, restrict and various alignments, they follow the type that they qualify. These attributes are not regarded as part of the type. If these attributes are applied to a derived type, they must follow outside the type angular brackets. Examples:

```
var %x f64 volatile align(16)        # %s is a f64 value that is volatile and aligned on 16
byte boundary
var %p <* f32> const volatile        # %p is a pointer to a f32 value, and %p itself is
const and volatile
```

Alignments are specified in units of bytes and must be power of 2. Alignment attributes must only be used to increase the natural alignments of the types, to make the alignments more stringent. For decreasing alignments, the generator of MAPLE IR must use smaller types to achieve the effect of packing instead of relying on the align attribute.

Incomplete Type Specification

Languages like Java allow contents of any object to be referenced without full definition of the object being available. Their full contents are to be resolved from additional input files in later compilation phases. MAPLE IR allows structs, classes and interfaces to be declared incompletely so their specific contents can be referenced. Instead of the *struct*, *class* and *interface* keywords, *structincomplete*, *classincomplete* and *interfaceincomplete* should be used instead.

Type Declaration

syntax: `type <id-name> <derived-type>`

Type names are also prefixed with either '$' or '%' based on whether the scope is global or local. Example:

```
type $i32ptr <* i32>          # the type $i32ptr is a pointer to i32
```

Primitive types are not allowed to be given a different type name.

Attributes are not allowed in type declaration.

Once a type name is defined, specifying the type name is equivalent to specifying the derived type that it stands for. Thus, the use of a type name should always be enclosed in angular brackets.

Java Class and Interface Declaration

A Java class designates more than a type, because the class name also carries the attributes declared for the class. Thus, we support declaration of Java classes via:

syntax: `javaclass <id-name> <class-type> <attributes>`

<id-name> must have '$' as prefix as class names always have global scope. For example:

```
javaclass $Puppy <class [{@color](mailto:%7B@color) i32}> public final         # a java
class named "Puppy" with a single field "color" and attributes public and final
```

A javaclass name should not be regarded as a type name as it contains additional attribute information. It cannot be enclosed in angular brackets as it cannot be referred to as a type.

A java interface has the same form as the class type, being able to extend another interface, but unlike class, an interface can extend multiple interfaces. Another difference from class is that an interface cannot be instantiated. Without instantiation, the data fields in interfaces are always allocated statically. For example,

```
interface <$interfaceA> {      #this interface extends interfaceA
  @s1 int32,                   # data fields inside interfaces are always statically
allocated
  &method1(int32) f32 }        # a method declaration
```

MAPLE IR handles an interface declaration as a type declaration. Thus, the above can be specified after the type keyword to be associated with a type name. Separately, the javainterface keyword declares the symbol associated with the interface:

syntax: `javainterface <id-name> <interface-type> <atteributes>`

<id-name> must have '$' as prefix as interface names always have global scope. For example:

```
javainterface $IFA <interface{&amethod(void) int32}> public static  # $IFA is an interface
with a single method &amethod
```

Again, a javainterface name should not be regarded as a type name, as it is a symbol name. When a class implements an interface, it specifies the javainterface name as part of its comma-separated contents, as in:

```
class <$PP> { &amethod(void) int32,      # this class extends the $PP class, and &amethod is
a member function of this class

           $IFA }                          # this class implements the $IFA interface
```

Function Declaration

syntax:

```
func <func-name> <attributes> (

                         var <parm0> <type>,

                              . . .

                         var <parmn> <type>)  <ret-type0>,  . . .
<ret_typen> {

                     <stmt0>

                       . . .

                      <stmtn> }
```

<attributes> provides various attributes about the function, like static,extern, etc. The opening parentheses starts the parameter declarations, which can be empty. Each <parmi> is of the form of a var or reg declaration declaring each incoming parameter. If the last parameter is specified as "...", it indicates the start of the variable part of the arguments as in C. Following the parameter declarations is a list of the multiple return types separated by commas. If there is no return value, <ret-type0> should specify void. Each <ret-typei> can be either primitive or derived. If no statement follows the parentheses, then it is just a prototype declaration.

funcsize

syntax: `funcsize <size-in-bytes>`

This directive appears inside the function block to give the Maple IR code size of the function.

framesize

syntax: `framesize <size-in-bytes>`

This directive appears inside the function block to give the stack frame size of the function.

moduleid

syntax: `moduleid <id>`

This directive appears inside the function to give the unique id of the module which the function belongs to.

upformalsize

syntax: `upformalsize <size-in-bytes>`

This directive appears inside the function block to give the size of upformal segment that stores the formal parameters being passed above the frame pointer %%FP.

formalwordstypetagged

syntax: `formalwordstypetagged = [ <word-values> ]`

This specifies a bitvector initialized to the value specified by the list of space-separated 32-bit integer constants. The Nth bit in this bitvector is set to 1 if the Nth word in the upformal segment has a type tag, in which case the type tag is at the (N+1)th word.

formalwordsrefcounted

syntax: `formalwordsrefcounted = [ <word-values> ]`

This specifies a bitvector initialized to the value specified by the list of space-separated 32-bit integer constants. The Nth bit in this bitvector is set to 1 if the Nth word in the upformal segment is a pointer to a reference-counted dynamically allocated memory block.

localwordstypetagged

syntax: `localwordstypetagged = [ <word-values> ]`

This specifies a bitvector initialized to the value specified by the list of space-separated 32-bit integer constants. The Nth bit in this bitvector is set to 1 if the -Nth word in the local stack frame has a type tag, in which case the type tag is at the (-N+1)th word.

localwordsrefcounted

syntax: `localwordsrefcounted = [ <word-values> ]`

This specifies a bitvector initialized to the value specified by the list of space-separated 32-bit integer constants. The Nth bit in this bitvector is set to 1 if the -Nth word in the local stack frame is a pointer to a reference-counted dynamically allocated memory block.

Initializations

When there are initializations associated with a var declaration, there is '=' after the var declaration, followed by the initialization value. For aggregates, the list of initialization values are enclosed by brackets '[' and ']', with the values separated by commas. In arrays, the initialization values for the array elements are listed one by one, and nested brackets must be used to correspond to elements in each lower-order dimension.

In specifying the initializations for a struct, inside the brackets, field-ID followed by '=' must be used to specify the value for each field explicitly. The fields' initialization values can be listed in arbitrary order. For nested structs, the usage of nested brackets is optional, because field-IDs at the top level can uniquely specify fields in nested structs. But if nested brackets are used, then field-ID usage within the nested brackets is relative to that corresponding level of sub-struct. Example:

```
type %SS <struct { @g1 f64, @g2 f64 }>
var %s [struct{@f1i32,
                      @f2 <%SS>,
                      @f3:4 i32} = [ 1 = 99,
        2 = [1= 10.0],          # field f2.g1 has field ID 1 in struct %SS and is
initialized to 10.0
        4 = 22.2,           # field f2.g2 has field ID 4 in struct %s and is initialized to
22.2
        5 = 15 ]               # field f3 (4 bits in size) has field ID 5 in struct %s
and is initialized to 15
```

Type Parameters

Also called generics, type parameters allow derived types and functions to be written without specifying the exact types in parts of their contents. The type parameters can be instantiated to different specific types later, thus enabling the code to be more widely applicable, promoting software re-use.

Type parameters and their instantiation can be handled completely by the language front-ends. MAPLE IR provides representation for generic types and generic functions and their instantiation so as to reduce the amount of work in the language front-ends. A MAPLE IR file with type parameters requires a front-end lowering phase to de-genericize the IR before the rest of the MAPLE IR components can process the code.

Type parameters are symbol names prefixed with "!", and can appear anywhere that a type can appear. Each type or function definition can have multiple type parameters, and each type parameter can appear more than one time. Since type parameters are also types, they can only appear inside the angular brackets "<" and ">", e.g. <!T>. When the definition of a derived type contains any type parameter, the type becomes a generic type. When the definition of a function contains any type parameter, the function becomes a generic function. A function prototype cannot contain generic type.

A generic type or generic function is marked with the *generic* attribute to make them easier to be identified.

A generic type or function is instantiated by assigning specific non-generic types to each of its type parameters. The instantiation is specified by a list of such assignments separated by commas. We refer to this as an instantiation vector, which is specified inside braces "{" and "}". In the case of the instantiation of a generic type, the type parameter is immediately followed by the instantiation vector. Example:

```
type $apair <struct {@f1 <!T>, @f2 <!T>}>    # $apair is a generic type

var $x <$apair{!T=f32}>           # the type of $x is $apair instantiated with f32 being
assigned to the type parameter !T
```

A generic function is instantiated by invoking it with an instantiation vector. The instantiation vector immediately follows the name of the generic function. Since the instantiation vector is regarded as type information, it is further enclosed inside the angular brackets "<" and ">". Invocation of generic functions must be via the opcodes **callinstant** and **callinstantassigned**, which correspond to **call** and **callassigned** respectively. Example:

```
 func &swap (var %x <!UU>, var %y <!UU>) void {         # &swap is a generic
 function to swap the contents of its two parameters

    var %z <!UU>

    dassign %z (dread agg %x)

    dassiign %x (dread agg %y)

    dassign %y (dread agg %z)
```

```
   return

}

   . . .

callinstant &swap<{!UU=<$apair{!T=i32}>}> (          # &swap is instantiated with type
argument <$apair{!T=i32}>, itself an instantiated type

 dread agg %a,                        # the actual parameters %a and %b must have same type as
the same instantiated type

 dread agg %b)
```

Examples

Example 1

Function definitions and arithmetic expressions:

C code:

```c
int foo(int i,int j){
  return(i + j)* -998;
}
```

Maple IR:

```
func &foo (var %i i32, var %j i32) i32 {
  return (
    mul i32 (
      add i32 (dread i32 %i, dread i32 %j),
      constval i32 -998))}
```

Example 2

Loops and arrays

C source:

```c
float a[10];
void init(void){
  int I;
  for(i=0; I<10; i++)
    a[i]=i*3;
}
```

Maple IR:

```
var $a <[10] f32>
func &init() void{
 var %i i32
 dassign %i(constval i32 0)
 while(
 lt i32 i32(dread i32 %i, constval i32 10)){
   iassign<*[10] f32>(
     array a32<*[10] f32>(addrof a32 $a, dread i32 %i),
     mul f32(dread i32 %i, constval i32 3))
    dassign %i(
     add i32(dread i32 %i, constval i32 1))}}
```

Example 3

Types and structs

C source:

```
typedef struct SS{
  int f1;
  char f2:6;
  char f3:2;
}SS;
SS foo(SS x){
  x.f2=33;
  return x;
}
```

Maple IR:

```
type $SS <struct {@f1 i32, @f2:6 i8,@f3:2 i8}>
func &foo (var %x <$SS>) <$SS> {
  dassign %x 2 (constval i32 32 )
  return (dread agg %x) }
```

Example 4

if-then-else, call and block

C source:

```
int fact(int n) {
  if(n!=1)
    return n*fact(n-1);
  else return 1;
}
```

Maple IR:

```
func &fact (var %n i32) i32 {
  if (ne i32 (dread i32 %n, constval i32 1)) {
    call $fact (sub i32 (dread i32 %n, constval i32 1))
    return (mul i32 (dread i32 %n, regread i32 %%retval))
  }
  return(constval i32 1)
}
```