CHAPTER 1

# WHIRL Intermediate Language Specification

## 1.1 Introduction

This document discusses WHIRL, the intermediate language (IR) for the SGI Pro64™ compiler. Using a common IR enables a compiler to support multiple languages and multiple processor targets. The different front-ends of the SGI Pro64 compiler translate the different languages to WHIRL. The SGI Pro64 compiler has a sophicated back-end composed of multiple components: the inter-procedural analyzer and optimizer (IPA), loop-nest optimizer (LNO), global scalar optimizer (WOPT) and code generator (CG). WHIRL serves as the common interface among all these components.

Adapting a common intermediate representation for as many phases of the compilation process as possible has numerous advantages. In the compilation process, some optimization passes like constant propagation, dead code elimination, and various liveness problems, have to be re-applied at different times and in different components of the compiler. With a common IR, a single implementation of an optimization pass is sufficient. Communication between the compilation phases is also easier, since they work under the same medium.

WHIRL is designed to support C, C++, Java, FORTRAN77 and FORTRAN90. It is expected that additional programming languages can be targeted to WHIRL without substantial difficulties.

This document is intended to be a clear, precise and complete specification of WHIRL. A compiler front-end vendor should be able to port their front-end to WHIRL based on this document and using the WHIRL software package. The generated WHIRL code should not assume any semantics other than what is specified in this document; otherwise it is considered incorrect WHIRL.

A separate document describes the symbol table portion of WHIRL.

## 1.2 Compilation Targets

WHIRL is designed to support effective compilation of program code to multiple target processor architectures. As such, the WHIRL generated by the front-ends does not assume specific target processor characteristics. Instead, it targets the abstract C machine that models the semantics of the C programming language. In particular, integers are promoted to either 32 or 64 bits beforechwwww being involved in computations.

As compilation proceeds, the code sequence at lower levels of WHIRL will more accurately reflect the target machine's support of program operations. At lower levels of WHIRL, the code generated is different for different target processor, because the representation is restricted to what is actually supported in the target ISA. This is necessary for the back-end to produce optimal code sequences for each target processor. More details are given in the next Section.

## 1.3 The Levels of WHIRL

Nowadays, optimization is an indispensible part of the compiler, and compiler back-ends have grown to become larger and more complicated. As we add to the classes of optimizations that the compiler has to perform, we are increasing the complexity of the compiler back-end at the same time. With each new optimization that we add, we have to be more concerned about the robustness of the compiler, because each new optimization is one more source of instability for the entire compiler. Thus, it is necessary to find ways to simplify each optimization without compromising on the quality of the output code. Since optimizations operate on IRs, it is important that we design the most efficient form of representation for each optimization phase to work on.

Compilation can be viewed as a process of gradual transition from the high level language constructs to the low level machine instructions. In between, there are different levels of IR possible. The closer an IR is to the source language, the higher is its level. The more an IR resembles the machine instructions, the lower is its level. Table 1 contrasts the general characteristics between high and low level representations.

**Table 1 Differences between high and low level representations**

| Characteristics | High level IR | Low level IR |
|---|---|---|
| kinds of constructs | many | few |
| length of code sequences | short | long |
| form | hierarchical | flat |

Theoretically speaking, all optimizations can be performed on the lowest level machine instructions, because any optimization effect has to filter down to and expressible in them. This is, however, undesirable because of the following reasons:

1. Information content — Since high level program representation resembles the way the original program was written, it provides the optimizer with more exact information about the program, thus allowing it to do a better job in optimizing the program.

2. Granularity — By matching the granularity of the program representation with the granularity of the constructs that each optimization phase manipulates, the optimization phases can be implemented with less effort and operate much more efficiently.

3. Variations — The optimizer has to deal with more possible variations in the code sequences that perform a given task in low level IR, making it harder to recognize specific program semantics.

In general, the higher the level of the IR, the more assumptions the optimization phase can make about its representation, thus allowing it to gather information more efficiently and streamline its work load. In light of this, our approach in SGI Pro64 is to implement each optimization at as high a level as possible without affecting the quality of its work.

We have defined WHIRL to be capable of representing any level of IR except the level that corresponds to the machine instructions. The SGI Pro64 back-end performs a complete repertoire of optimizations. We define different levels of WHIRL, and define each optimization phase to work at a specific level of WHIRL. The front-ends generates the highest level of WHIRL. Optimization proceeds together with the process of *continuous lowering*, in which a WHIRL *lowerer* is called to translate WHIRL from the current level to the next lower level. At the end, the code generator translates the lowest level of WHIRL to its own internal representation that matches the target machine instructions. WHIRL thus serves as the common IR interface among all the back-end components. Because lowering is done gradually, a secondary benefit is that each lowering step is simpler and easier.

Figure 1 illustrates the relationships between the compilation process and the levels of WHIRL. The following subsections give the main characteristics of each WHIRL level. In the specification of each WHIRL operator later in this document, the levels where the operator is allowed to exist are specified.

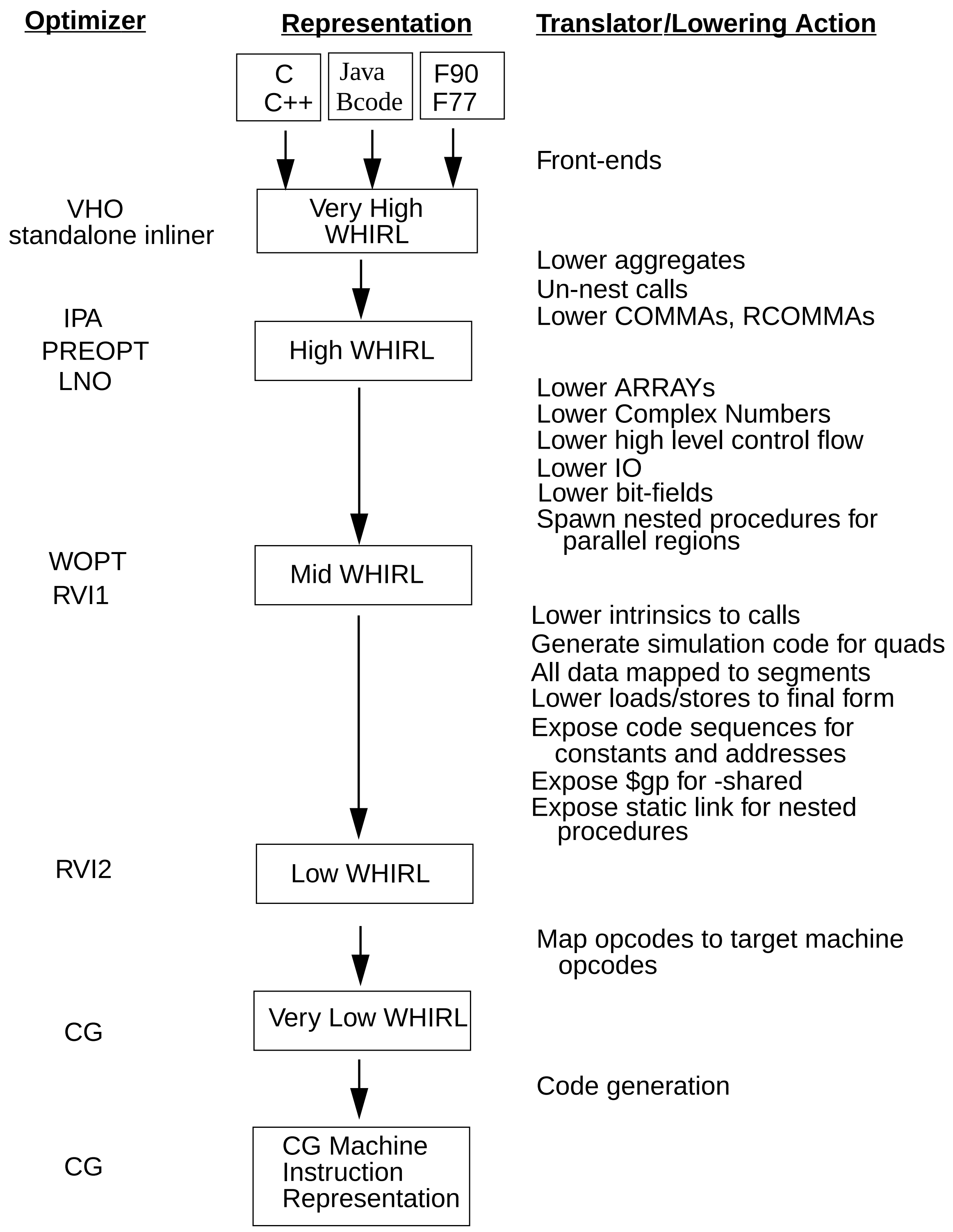
Optimizer
Representation
Translator/Lowering Action
C
C++
Java
Bcode
F90
F77
Front-ends
VHO
standalone inliner
Very High WHIRL
Lower aggregates
Un-nest calls
Lower COMMAs, RCOMMAs
IPA
PREOPT
LNO
High WHIRL
Lower ARRAYs
Lower Complex Numbers
Lower high level control flow
Lower IO
Lower bit-fields
Spawn nested procedures for parallel regions
WOPT
RVI1
Mid WHIRL
Lower intrinsics to calls
Generate simulation code for quads
All data mapped to segments
Lower loads/stores to final form
Expose code sequences for constants and addresses
Expose $gp for -shared
Expose static link for nested procedures
RVI2
Low WHIRL
Map opcodes to target machine opcodes
CG
Very Low WHIRL
Code generation
CG
CG Machine Instruction Representation

**Figure 1 Continuous Lowering in the SGI Pro64 Compiler**

### 1.3.1 Very High (VH) WHIRL

This level of WHIRL is output by the front-ends, and faithfully corresponds to the structure of the program in the source code. Optimizations performed on this level of WHIRL can be perceived as optimizing with respect to the programming language constructs. This level of WHIRL can be translated back to C and FORTRAN source code with only minor loss of semantics. The tools **whirl2c**, **whirl2f** and **whirl2f90** are provided for this purpose.

In this level of WHIRL, calls are allowed to be nested. The **COMMA**, **RCOMMA** and **CSELECT** operators are allowed. The operators related to FORTRAN90 aggregates **TRIPLET**, **ARRAYEXP**, **ARRSECTION** and **WHERE** are allowed. These constructs are not allowed in lower levels of WHIRL.

The Very High WHIRL Optimizer (VHO) and standalone inliner work on VH WHIRL.

### 1.3.2 High (H) WHIRL

At this level of WHIRL, side effects can only occur at statement boundaries, and control flows are fixed. As a result, procedure calls are not allowed to be nested, as are statements nested via the **COMMA** and **RCOMMA** operators. This level of WHIRL can also be translated back to C and FORTRAN source code, though not to very close correspondence to the original source.

In this level of WHIRL, high level control flow constructs are preserved via the operators **DO_LOOP**, **DO_WHILE**, **WHILE_DO**, **IF**, **CAND** and **CIOR**. The form of FORTRAN I/O statements are preserved via **IO** and **IO_ITEM**. The form of array subscripting is preserved via **ARRAY.** Bit-field accesses can be represented in high-level form via field-id.

IPA, LNO and the PREOPT part of the global scalar optimizer operate in H WHIRL. Pseudo-registers can be generated by the compilers to store values. Integer pseudo-registers must be of either 32- or 64- bit sizes.

### 1.3.3 Mid (M) WHIRL

At this level of WHIRL, the representation starts to reflect the characteristics of the target ISA. In general, for maximum optimization effectiveness, each WHIRL instruction should map to one instruction in the target ISA. A WHIRL instruction that is no-op in the target processor should not be generated. The WHIRL code sequence should correspond to the final generated code sequence in the target ISA. Pseudo-registers are assumed to be of sizes corresponding to the sizes of the machine registers, but if their sizes in WHIRL are smaller, CG can allocate the smaller spill locations when spilling them. Physical registers also start to show up at this level of WHIRL. Data type B can start to show up at this level of WHIRL if the target provides predicate registers.

At this level of WHIRL, control flow must be uniformly represented via **TRUEBR**, **FALSEBR**, **GOTO** or **COMPGOTO**. **IO** must have been lowered to calls. **ARRAY** must have been lowered to address expressions. Bit-field accesses must be represented via **LDBITS, STBITS, ILDBITS and ISTBITS,** and then furthered lowered to **EXTRACT_BITS** and **COMPOSE_BITS.** Such uniform code generation strategies allow common code sequences to be identified during optimization.

The global scalar optimizer WOPT works on M WHIRL.

### 1.3.4 Low (L) WHIRL

WOPT performs two rounds of register variable identification (RVI). The first round is performed on M WHIRL. The purpose of L WHIRL is to expose candidates for the second round of RVI.

At L WHIRL, **LDID** and **STID** are lowered into **ILOAD** and **ISTORE** so that the base address is exposed to RVI, while **ILOAD** and **ISTORE** map to the load and store instructions in the target ISA. Constants, including **LDA**s, are lowered into the exact code sequence in which they are generated in the target ISA. Calls is lowered to **PICCALL** under -shared compilation. **COMPGOTO** is lowered to **XGOTO**.

### 1.3.5 Very Low (VL) WHIRL

This is the lowest level of WHIRL before translation to CG's machine instruction representation. It exhibits strict one-to-one correspondence with the target machine instructions. As a result, the generated instruction mix is very target-dependent.

VL WHIRL only exists internal to CG. Some peephole optimizations are performed on VL WHIRL.

## 1.4 The Components of WHIRL

A WHIRL file generated by the front-end consists of WHIRL instructions and WHIRL symbol tables. A separate docment describes the structure of the WHIRL symbol tables. WHIRL instructions contain references to the symbol tables via fields that are ST_IDX and TY_IDX.

The instruction part of the WHIRL file represents the program code, organized in program units (PUs). The WHIRL instructions are linked up in strictly tree form, and we refer to each node in the tree as a WHIRL node. DAGs are not allowed. The same WHIRL tree is used to represent both control flow and expressions. Each PU is a single tree.

We now describe the content of the WHIRL node.

### 1.4.1 Operators

The **operator** field in a WHIRL node specifies the operation performed by the instruction.  Operators in WHIRL can be divided into three categories: structured control flow, statements, and expression. These are represented hierarchically in the tree. It is illegal for a structured control flow operator to be a descendant of a different type of operator. Similarly, a statement cannot be a descendant of an expression. Statements have the further restriction that they cannot be nested, i.e. a statement cannot be a descendant of another statement. There are, however, exceptions to these rules in VH and H WHIRL.

### 1.4.2 Result and Descriptor Types

The operation specified the WHIRL operator can be further qualified by the result type (**res**) and descriptor type (**desc**). **res** gives the data type of the result of the operation, while **desc** gives the data type of the operands.

**operator** together with **res** and **desc** fully specifies an operation. It should not be necessary to examine the kids of the node in order to determine the exact operation to be performed.

### 1.4.3 Supported Data Types

The following data types are supported in WHIRL:

| | |
|---|---|
| B | boolean (value is either 0 or 1) |
| I1 | 8-bit signed integer. |
| I 2 | 16-bit signed integer. |
| I4 | 32-bit signed integer. |
| I8 | 64-bit signed integer. |

| | |
|---|---|
| U1 | 8-bit unsigned integer. |
| U2 | 16-bit unsigned integer. |
| U4 | 32-bit unsigned integer. |
| U8 | 64-bit unsigned integer. |
| A4 | 32-bit address (behaves as unsigned). |
| A8 | 64-bit address (behaves as unsigned). |
| F4 | 32-bit IEEE floating point. |
| F8 | 64-bit IEEE floating point. |
| F10 | 80-bit IEEE floating point. |
| F16 | 128-bit IEEE floating point. |
| FQ | 128-bit SGI floating point. |
| C4 | 32-bit complex (64 bits total). |
| C8 | 64-bit complex (128 bits total). |
| CQ | 128-bit complex (256 bits total). |
| V | Void. |
| M | struct. |
| BS | bits. |

Type B corresponds to predicate registers, and is useful only if the target has such registers; it is introduced into the compilation starting in M WHIRL by the global optimizer (WOPT). Booleans are represented as integer types otherwise.

The I1, I2, U1, U2 and BS data types are allowed only in the **desc** field of memory access operations.

Type A4 and A8 gives the information that the integer value specifies an address, thus allowing the optimizer to perform more aggressive optimizations. It behaves as unsigned, in the sense that, if there is a choice, it will be zero-extended instead of sign-extended.

Type FQ is currently supported in software only, and is lowered to F8 in L WHIRL. The complex types are included because they allow the loop nest optimizer to perform analysis of programs with complex arrays more efficiently. The complex types are lowered to the floating point types in M WHIRL.

Type M indicates a value made up of composite fields. Type M is not allowed in arithmetic operations. When a type field is unused for an operator, it should be initialized to V.

In the specification of the WHIRL opcodes, we give the allowed types for **res** and **desc** for each operator. We'll use the following lower case letters to specify groups of data types:

| | |
|---|---|
| i | Any of I4,I8,U4,U8,A4,A8 integral types |
| f | Any of F4,F8,F10,F16,FQ floating point types |
| z | Any of C4,C8,CQ complex types |

### 1.4.4 Kid Pointers

WHIRL nodes other than **BLOCK** that are non-leaves contain pointers to their children in the **kids** array. For operators that have a variable number of kids, field **kid_count** gives the number of children. **BLOCK** nodes contain **first** and **last** pointers to a doubly linked list of statements.

### 1.4.5 Next and Previous Pointers

The children of a **BLOCK** node must be statement nodes, and statement nodes all have **next** and **previous** pointers which link them together. These fields are NULL for any statement nodes that are not children of **BLOCK**s. The first statement of a **BLOCK** has null **previous** field, and the last statement has null **next** field.

### 1.4.6 Offset

All load and store opcodes have **offset** fields. The load-address opcode **LDA** also uses the offset field to specify the exact address to load. In the case of the indirect load and store opcodes, there may be code to compute addresses prior to the loads and stores. In VH and H WHIRL,it is not legal to fold the offset fields in either the load and store opcodes or **LDA** into the address computation. Doing this will impact the ability of the loop nest optimizer to do data dependence analysis.

The **offset** field is used to keep other contents for other operators.

### 1.4.7 Mapping Mechanism

Different phases of the compiler may need to store additional information associated with individual whirl nodes. Rather than providing a pointer in each tree node for every conceivable data structure, WHIRL provides a general mapping, or annotation, mechanism. One can view this mechanism as a mapping table (although the actual implementation may be quite different). Each node contains a word-sized **map_id** that effectively maps to a row in the table. By creating a new map, the user reserves a column in the table. The user can then enter or query a value for any map for any WHIRL node in constant time.

As an example, imagine that a compiler pass wishes to store a parent pointer for every control flow node in the tree. The pass would call

```
parent_map = WN_MAP_Create(mempool).
```

At this point, parent_map would contain the name of a new mapping. Memory to store information about the mapping will be allocated from mempool. The pass would then visit every control flow node, nd, in the tree, calling

```
WN_MAP_Set(parent_map, wn, parent).
```

Now, the parent of any control flow node, nd, can be found by calling

```
parent = WN_MAP_Get(parent_map, wn).
```

To avoid creating too many entries that are unused, the WHIRL nodes are divided into different categories according to the operator. Assigned map IDs are unique only within each category. There is one category for all the structured control flow statements, one for all the load and store nodes, one for **ARRAY** nodes, one for all other statement nodes, and one for all other expression nodes. Map IDs are also unique only within each PU, and the map tables are organized on a per-PU basis in the WHIRL file.

### 1.4.8 Source Position Information

The 64-bit field **linenum** for specifying source position information is allocated only for statement nodes. The line number is stored in a 32-bit field. The remaining 32 bits contain the file and column number.

### 1.4.9 Additional Fields

There are other operator-specific fields such as symbol table indices and type table indices. These fields are underlined and described in the operator specifications.

## 1.4.10 WHIRL Node Layout

A WHIRL node is represented by the struct WN. The minimum allocated size of struct WN is 24 bytes., which include pointers to two kids. If the node has more than two kids, the struct is extended at the end for the additional kid pointers needed. If the node is a statement, four additional words are allocated before the struct for **linenum** and the **previous** and **next** pointers. Table 2 gives the layout of struct WN.

**Table 2 Layout of the WHIRL node**

| Offset | Field | Description | Field size |
|---|---|---|---|
| byte -16 | prev | previous pointer | word |
| byte -12 | next | next pointer | word |
| byte -8 | linenum | source position information | double word |
| byte 0 | offset | offset for loads, stores, LDA, IDNAME;<br>no, of entries, COMPGOTO and SWITCH;<br>length in bits for CVTL;<br>label number;<br>flags for calls, PARM and REGION;<br>break code for TRAP, ASSERT; | word |
| byte 0 | trip_est | estimated trip count for LOOP_INFO; | half-word |
| byte 2 | depth | loop nesting depth for LOOP_INFO; | half-word |
| byte 4 | st_idx | symbol table index;<br>type index for all except LDA, LDID, STID;<br>last label for COMPGOTO and SWITCH;<br>number of exits for REGION;<br>id for intrinsics;<br>flags field for PREFETCH, LOOP_INFO ;<br>region supplement: EXC_SCOPE_BEGIN; | word |
| byte 0 | elem_size | element size for ARRAY; | double word |
| byte 8 | operator | WHIRL operator; | byte |
| byte 9 bit 0 | res | result type; | 5 bits |
| byte 9 bit 5 | kid_count | number of kids for n-ary operators;<br>field ID for operators with fixed no. of kids;<br>bit_offset at most significant 7 bits and<br>bit_size at least significant 7 bits for<br>LDBITS, STBITS, ILDBITS, ISTBITS,<br>EXTRACT_BITS and COMPOSE_BITS; | 14 bits |
| byte 11 bit 3 | desc | descriptor type; | 5 bits |
| byte 12 | map_id | index into map table; | word |

**Table 2 Layout of the WHIRL node**

| Offset | Field | Description | Field size |
|---|---|---|---|
| byte 16 | kids[0] | kid 0;<br>first pointer for BLOCK;<br>flags for LABEL; | word |
| byte 20 | kids[1] | kid 1;<br>type index for LDA, LDID, STID;<br>address type pointer for ILOAD;<br>last pointer for BLOCK; | word |
| byte 16 | const_val | 64-bit integer constant; | double word |
| byte 24+n | kids[2+n] | the *(2+n)th* kid for n ≥ 0; | word |

In the upcoming operator specification in this document, any fields other than operator, prev/next, linenum and kid_count that an operator use will be underlined so that the reader can know at a glance what additional fields in the node are used for each operator.

## 1.5 Structured Control Flow Statements

Structured control flow statements in WHIRL are hierarchical in nature. All the statements in a particular control flow structure are descendents of the node representing that structure. All the control flow opcodes have a 'V' in their result type and descriptor fields. Except **FUNC_ENTRY** and **BLOCK**, structured control flow opcodes are not allowed in M—VL WHIRL. All of these opcodes use the prev and next fields.

- **FUNC_ENTRY** **[VH—VL]**
  This operator represents a function entry. This operator will be at the top of every tree. st_idx points to the name of the procedure or function. Kids 0..n-4 are **IDNAME** leaves containing the names of the formal parameters. Kid n-3 is a **BLOCK** node containing a list of **PRAGMA**s that are relevant to the compilation of the PU. Kid n-2 is a **BLOCK** containing a list of **PRAGMA**s that are relevant to the compilation at the call sites of the PU. For a nested PU, this pragma list must be present to identify any non-local variables accessed in the PU to ensure correct compilation at the call sites. Kid n-1 is a **BLOCK** node giving the body of the procedure.

- **BLOCK** **[VH—VL]**
  This operator represents a list of subtrees. It contains an arbitrary number of children connected together via a doubly linked list, and pointed to by the first and last fields. The prev field of the first child and the next field

of the last child must be null. It is the only operator for which the number of children is not fixed at node creation time.The kid_count field is undefined for this operator. A **BLOCK** may not be the direct child of another **BLOCK**. An empty **BLOCK** is allowed, in which case the head of the doubly linked list is null. In M—VL WHIRL, this operator can only appears under **FUNC_ENTRY**.

- **REGION** **[H—VL]**
  This operator specifies a nested sub-region. The region flags field specifies the WHIRL level in the region. It has three kids, all of which must be **BLOCK**s. The number-exits field gives the number of exit points from the region. Kid 0 is a **BLOCK** that defines a jump table by its list of **REGION_EXIT**s. The number of **REGION_EXIT**s must be equal to the number of exits. Kid 1 gives a list of **PRAGMA**s that affect (and only affect) the compilation of the region. Kid 2 gives the content of the region. A region serves as a unit of compilation. Regions can be nested one inside another. The outermost region is the block corresponding to **FUNC_ENTRY**. WHIRL level changes are allowed only at region boundaries. When the current compilation unit contains **REGION** nodes, they are to be treated as black boxes while working on the current compilation unit. **REGION** nodes cannot contain references and definitions of pseudo-registers that are live-in or live-out with respect to the node. Values can be passed in and out of the black boxes via dedicated registers at the region boundaries. The WHIRL level of a region must be lower than or equal to the level of its enclosing region. A compilation component may choose to ignore a region boundary at which the WHIRL level does not change, in which case it will optimize the code of the region together with the enclosing region. In general, nested regions should be compiled inside-out. An additional use of this node is to specify a region to be parallelized. In the course of compilation, a segment of code to be parallelized is first marked as a parallel region. The lowering process will spawn off the region as a nested procedure that will be called via synchronization routines during parallel execution.

- **DO_LOOP** **[VH—H]**
  This operator has the semantics of a Fortran Do loop. Kid 0 is an **IDNAME** representing the index variable, which must be of type integer. Kid 1 must be an **STID** statement initializing the index variable, which must not be null. Kid 2 is a comparison expression for the end condition. The comparison must use **GE**, **GT**, **LE** or **LT**, and any content other than the induction variable in this expression must be loop invariant. Kid 3 must be an **STID** statement that increments the index variable via an **ADD** by a step amount. The step must be an expression that is loop invariant. Kid 4 is a

**BLOCK** node representing the body of the do loop. If Kid 5 is present, it must be a **LOOP_INFO** that gives additional information about the loop.

- **DO_WHILE** **[VH—H]**
  A while loop. Kid 0 is a boolean expression. Kid 1 is a **BLOCK** node representing the block of statements that is executed while kid 0 returns non-zero. The condition is tested at the end of the loop, so the block is executed at least once.

- **WHILE_DO** **[VH—H]**
  A while loop. Kid 0 is a boolean expression. Kid 1 is a **BLOCK** representing the block of statements that is executed while Kid 0 returns non-zero. The condition is tested at the start of the loop.

- **IF** **[VH—H]**
  This operator represents a structured logical if statement. Kid 0 is an expression, and both kids 1 and 2 must be **BLOCK**s. Kid 1gives a block of statements that is executed if Kid 0 evaluates to some non-zero value. Kid 2 gives another block of statements that is executed if Kid 0 evaluates to zero. If this statement has no else part, the block for Kid 2 has an empty statement list. The flags field is used to provide compilation-related information for this node.

**DO_LOOP**, **DO_WHILE**, **WHILE_DO** and **IF** represent only well-formed high-level control constructs. The blocks associated with them cannot be the target of jumps from outside. To make it easier for the front-ends, we do tolerate illegal high-level control constructs in the front-ends' output. Such illegal high-level control constructs will be screened out and converted to use ordinary control flow constructs by the first optimization phase.

## 1.6 Other Control Flow Statements

This section describes the remaining control flow statements in WHIRL, which are not hierarchical. They are allowed at all levels of WHIRL. All of these operators use the prev and next fields.

- **GOTO** **[VH—VL]**
  An unconditional branch to the label in the current procedure as given by label_number.

- **GOTO_OUTER_BLOCK** **[VH—VL]**
  An unconditional branch from a nested procedure to the label in a parent procedure as given by label_number. It involves unwinding of the procedure call stack.

- **SWITCH** **[VH]**
  A switch statement in a form close to the source code. An internal field, number_entries, gives the number of cases in the jump table. Another field, last_label, gives the label that marks the end of the code compiled from the switch statement in the source program. Kid 0 is the switch expression, which must be of type integer. Kid 1 is a **BLOCK** that defines the jump table by a list of **CASEGOTOs**, the number of which equals number_entries. Kid 2 is a **GOTO** giving the default jump target. If there is no default target (i.e. the front-end guarantees that a match case can be found), then Kid 2 does not exist. This statement will be lowered to the control flow constructs that most efficiently implement the switch.

- **CASEGOTO** **[VH]**
  This is used only within a **SWITCH** to specify jump targets for individual case values. The const_val field gives the integer case value. The label_number field gives the target of the jump if the switch expression evaluates to the given case value.

- **COMPGOTO** **[VH—M]**
  A non-structured computed goto statement. An internal field, number_entries, gives the number of entries in the jump table. Another field, last_label, gives the label that marks the end of the code compiled from the switch statement in the source program; a value of 0 means no information, and is used in the case of a FORTRAN computed/assigned goto, in which the jump targets are not contiguous. Kid 0 is the switch value, and must evaluate to a 0-based integer index. Kid 1 is a **BLOCK** that defines the jump table by its list of **GOTO**'s. The number of **GOTO** nodes must equal number_entries. For index value 0, the first **GOTO** is executed; for the next index value, the next **GOTO** is executed, etc. Kid 2 is a **GOTO** giving the default jump target. If there is no default target (i.e. the front-end guarantees that the switch value is in range), then Kid 2 does not exist.

- **XGOTO** **[L—VL]**
  This is formed out of lowering a **COMPGOTO**. st_idx gives the symbol table entry of the allocated jump table. Kid 0 is an expression that evaluates to the address to be jumped to, starting with the base address of the allocated jump table. Kid 1 is the same as Kid 1 in **COMPGOTO**. Number-entries gives the number of entries in the jump table as in **COMPGOTO**.

There is no default jump target. The default jump target in the original **COMPGOTO** must be handled by additional code generated during lowering.

- **AGOTO** **[VH—VL]**
  An assigned or indirect unconditional branch. The flow of control is transferred to the address evaluated by Kid 0.

- **REGION_EXIT** **[VH—VL]**
  This must exist within a **REGION** block, and specifies an exit out of the region. The label number specifies the label outside the region that the flow of control will transfer to. Exit out of a region can only be effected by executing this statement, and fall-through out of a region is not allowed. Other jump statements in the region must have their targets located inside the region.

- **ALTENTRY** **[VH—VL]**
  An alternate entry for the function, as translated from multiple entry subroutines in Fortran. st_idx names the entry point. Kid 0..n-1 are **IDNAME** leaves as in **FUNC_ENTRY**. However, there is no **BLOCK**, and control flows to the next statement. The code that appears before this operator must always ends with a **GOTO** to jump around the alternate entry, because the prolog code generated from lowering **ALTENTRY** is not to be executed unless control is entered via the alternate entry point.

- **TRUEBR** **[VH—VL]**
  A non-structured conditional branch. This node contains a label_number. Kid 0 is an expression that must evaluate to an integral value. If it evaluates to non-zero, control is transferred to the previously mentioned label. Otherwise, control flows to the next statement.

- **FALSEBR** **[VH—VL]**

  A non-structured conditional branch. This node contains a label_number. Kid 0 is an expression that must evaluate to an integral value. If it evaluates to zero, control is transferred to the previously mentioned label. Otherwise, control flows to the next statement.

- **RETURN** **[VH—VL]**
  Return from this procedure. There can be any number of return statements in a program unit. If a value is being returned, **RETURN_VAL** must be used instead. All return points must be explicitly specified via **RETURN** or **RETURN_VAL** even if it is the end of the function body.

- **RETURN_VAL res=any [VH—H]**
  Return from this function with the return value specified by Kid 0. This is lowered to **RETURN** with associated store statements in M WHIRL.

- **LABEL [VH—VL]**
  Define a label. This node contains a label_number. Any branch to the label will transfer control to the statement following this one. A flags field gives attributes about the label. In particular, one bit specifies that the label marks the start of an exception handler, in which case the label has to be treated as an entry point to the program unit. A **LOOP_INFO** may be attached to this node as Kid 0. Otherwise, Kid 0 must be NULL.

- **LOOP_INFO [H—VL]**
  Not a statement node, but exists as a kid of **DO_LOOP** in H WHIRL and **LABEL** otherwise. It provides information about a loop and does not translate into any executable code. In the case of being attached to an **LABEL**, it specifies the label as marking the start of the loop body, and the actual extent of the loop can be determined by finding all the basic blocks dominated by the label up to a branch back to that same label. The trip_est field is a 16-bit field that gives the estimated trip count of the loop; if it is larger than 16-bits, it should be represented as a large 16-bit number instead of being truncated; if 0, the information is not provided. The depth field gives the loop nesting depth of the content of the loop. The flags field provides various information about the loop, like innermost, loop wind-down, etc. Kid 0 must be an **LDID** that gives the induction variable of the loop. If Kid 0 is NULL, the loop has no induction variable. Kid 1 is an expression that evaluates to the exact trip count of the loop. If Kid 1 is NULL, the exact trip count cannot be specified or is unknown, as in the case of a **WHILE_DO**. If Kid 0 is NULL, Kid 1 must also be NULL. The trip count expression is for information only, and does not need to be optimized, since it replicates the executable code elsewhere that computes the trip count.

## 1.7 Calls

Because function calls can incur side effects, they are classified as statements rather than expression trees. Programming languages allow arbitrary nesting of function calls inside expressions. In VH WHIRL, those nestings are preserved by allowing call statements as nodes in an expression. The lowerer to H WHIRL has to unnest calls from expression trees in order to obey H WHIRL semantics. This also includes flattening out nested calls. Calls unnested from an expression need to be generated sequentially, and their return values need to be stored in pseudo-registers (pregs).

In VH and H WHIRL, return values from calls reside in the special pseudo-register specified by preg -1. This conforms to C language convention, in which only a single item can be returned, though it may be a composite item.

In lowering to M WHIRL, the actual return mechanism conforming to the target ISA and ABI is manifested. The actual return mechanism may involve multiple registers specified by dedicated pregs. The actual return mechanism may also create an implicit parameter that points to the memory block designated by the caller for returning a struct. **res** in the call node indicates the return type. Type V must be used for **res** if there is no subsequent read of the return pregs.

The code to read the return values in the pregs must be in the statements immediately after the call. If there is one return value, it must be in the first statement after the call. If there are *n* return values, it must be in the first *n* statements immediately after the call. In VH WHIRL, the statement that reads the return value can be a **COMMA**. Otherwise, the statement that reads the return value must be a simple **STID** or **ISTORE** whose right-hand-side contains only the **LDID** node of the return preg.

The WHIRL **ASM_STMT** is provided to support inline assembler instructions embedded in C code. Input operands to the asm are specified by **ASM_INPUT** kids of the **ASM_STMT**. Execution of **ASM_STMT** can result in the assignment of values to multiple output operands. The effect is represented by separate store statements that follow the **ASM_STMT**. The right-hand-sides of these stores refer to respective output operands via pregs with unique negative preg numbers. The correspondence of these pregs to the output operands are specified in Kid 1 of the **ASM_STMT**.

- **CALL** **res=any** **[VH—VL]**
  A direct call statement. st_idx gives the name of the procedure being called. Kids 0..n-1 are **PARM** nodes that specify the actual parameters to the call. WHIRL follows the C pass-by-value semantics. When **res** is not V, the return value is placed in one or more pregs; if more than one preg are used, **res** gives the data type in each preg. WHIRL follows the C pass-by-value semantics. A flags field gives attributes about the call that are useful for optimization around the call. The attributes are: *non_data_mod* (the called function modifies a data item that is not represented in the program), *non_parm_mod* (the called function modifies a non-local data item whose address is not passed as parameter), *parm_mod* (the called

```
        PARM
         ..
         ..
        PARM
              LDID <field_id for vptr>
            ILOAD  <field_id>
             ..
             ..
           ARRAY
         ILOAD
       ILOAD  <field_id>
     VFCALL
```

**Figure 2 Form for VFCALL**

function modifies a data item whose address is passed as parameter), *non_data_ref* (the called function references a data item that is not represented in the program), *non_parm_ref* (the called function references a non-local data item whose address is not passed as parameter), *parm_ref* (the called function references a data item whose address is passed as parameter), and *never_return* (the called function will cause control to exit the current program unit).

- **ICALL** **res=any** **[VH—VL]**
  An indirect call statement. Kid n-1 is the address of the procedure being called. Kids 0..n-2 are **PARM** nodes that specify the actual parameters to the call. WHIRL follows the C pass-by-value semantics. When **res** is not V, the return value is placed in one or more pregs; if more than one preg are used, **res** gives the data type in each preg. This operator contains a ty_idx, which gives the type information from the prototype definition of the function pointer. A flags field gives attributes about the call that are useful for optimization around the call.

- **VFCALL** **res=any** **[VH—H]**
  A virtual function call statement. Similar to **ICALL**, except that kid n-1 must be of the restricted form as given by Figure 2.

- **PICCALL** **res=any** **[L—VL]**
  A position-independent call statement, formed out of lowering a **CALL** under -shared compilation. Kid n-1 is the address of the procedure being called. Kids 0..n-2 are **PARM** nodes that specify the actual parameters to the call. When **res** is not V, the return value is placed in one or more

pregs; if more than one preg are used, **res** gives the data type in each preg. This operator contains the same st_idx as in the original **CALL**. A flags field gives attributes about the call that are useful for optimization around the call.

- **INTRINSIC_CALL res=any [VH—M]**
  A call to the intrinsic specified by the intrinsic field. Kids 0..n-1 are **PARM** nodes that specify the actual parameters to the call. When **res** is not V, the return value is placed in one or more negative pregs; if more than one preg are used, **res** gives the data type in each preg. A flags field gives attributes about the intrinsic that are useful for optimization around the intrinsic. Depending on the intrinsic and compilation options, it will either become a call or a sequence of instructions after it is lowered to L WHIRL.

- **IO [VH—H]**
  A call to the FORTRAN I/O intrinsic specified by the intrinsic field. This operator directly corresponds to an I/O statement in the FORTRAN source, and the trees underneath it also matches the I/O statement syntax, so as to allow easy translation back to FORTRAN source code by **whirl2f**. Kids 0..n-1 are all **IO_ITEM** nodes that specify the parameters in the I/O statement. Calls do not need to be unnested underneath an **IO**. Due to the need to tolerate such special semantics, the optimizations performed on the contents of this statement are limited and not as effective. There can be hidden references and side effects to program variables in this statement; to maintain proper optimization semantics, the hidden references and side effects must not be to any pseudo-registers, since their addresses cannot be taken. A flags field gives attributes about the intrinsic. In M WHIRL, this operator will be converted to a sequence of calls to the actual library routines.

- **ASM_STMT [VH—VL]**
  An inline assembler string. St_idx gives a CLASS_NAME symbol table entry whose name is the assembly code string. Kid 0 is a **BLOCK** containing a list of **PRAGMA**s and/or **XPRAGMA**s of type ASM_CLOBBER that indicate registers clobbered by the given assembly code. Kid 1 is a **BLOCK** containing a list of PRAGMAs of type ASM_CONSTRAINT, each of which indicates an operand constraint for an output operand and the negative preg number that will be used to refer to the output value corresponding to it. The code to actually transfer the output values to the output operands are generated as store statements that follow the **ASM_STMT**. These stores do not have to immediately follow the **ASM_STMT**. Because they may be arbitrarily separated from it, each negative preg used in an **ASM_STMT** must be unique (i.e. used only once)

within the program unit. From Kid 2 onwards are **ASM_INPUT** nodes, each giving an input operand expression and the corresponding constraint string. A flags field gives attributes about the **ASM_STMT**.

## 1.8 Other Statements

This section describes the WHIRL statements that are neither control flow nor stores. Store statements are described in the Memory Access Section. All statement operators use the prev and next fields.

- **EVAL** **[VH—VL]**
  The expression in Kid 0 is evaluated. This is used to force evaluation of an expression that does not produce a side effect. It is necessary for things like volatiles. If the expression does not have side effect, this statement can be optimized away.

- **PRAGMA** **[VH—VL]**
  This operator provides compilation directives for the current point of the program. The offset field gives the name of the pragma. st_idx, if not 0, gives the symbol associated with the directive. Additional values associated with the pragma are stored in the const_val field. The mapping mechanism can be used to store even more information for the pragma.

- **XPRAGMA** **[VH—VL]**
  This operator provides compilation directives like **PRAGMA**, but the directives are specified with respect to a WHIRL expression tree given by Kid 0 of this statement node. The number of kids must be 1. The offset field gives the name of the pragma. st_idx, if not 0, gives the symbol associated with the directive.

- **PREFETCH** **[H—VL]**
  This statement is generated by the front-end from a manual prefetch pragma, or automatically by LNO. Kid 0 computes an address which is added to the offset field. The optimizer needs not do anything to this operation other than optimizing the address computation. The flags field contains hints, which CG will incorporate into the prefetch instruction in the target machine code. The manual prefetch bit of the flags field identifies prefetches generated by the front-end that have not yet been processed by LNO, and thus can be ignored or deleted by the back-end phases when LNO is not run.

- **PREFETCHX** **[M—VL]**
  This operator is converted from **PREFETCH** by WOPT. It contains two kids, both of which must be **LDID**s corresponding to two pseudo-registers. The sum of the two kids give the computed address. The flags field contains hints, which CG will incorporate into the prefetch instruction in the target machine code.

- **COMMENT** **[VH—VL]**
  This operator does not translate into any executable code. It gives an ascii string for commenting purpose only. The st_idx field gives a CLASS_NAME symbol table entry whose name is the content of the comment.

- **TRAP** **[VH—VL]**
  When executed, this statement causes a breakpoint trap to occur. This operator is translated to either a instruction that causes a break, or a call to a runtime routine that eventually traps. The offset field contains the break code that specifies how the trap will be effected. Execution will not continue into the next statement. It can have a variable number of kids depending on the break code.

- **ASSERT** **[VH—VL]**
  This statement WHIRL node asserts the condition specified by Kid 0. If the result is true, nothing will happen. Otherwise, the effect is the same as executing the corresponding **TRAP**. The offset field contains the break code as in **TRAP**. This operator can be used to implement bounds-checking or assertions. It can have a variable number of kids depending on the break code. The compiler can delete this statement or generate **TRAP** if it can prove that the condition evaluates to true or false respectively.

- **AFFIRM** **[VH—VL]**
  This statement WHIRL node does not cause any executable code to be generated. It affirms that the condition specified by Kid 0 is always true, and that the compiler can take advantage of the information in performing optimizations.

- **FORWARD_BARRIER** **[VH—VL]**
  This operator designates a barrier to the code movement of memory access instructions in the forward direction (along the flow of control), used for MP support. If there is zero kid, all memory objects are affected. Otherwise, a *named barrier* is specified, and only the memory accesses represented by dereferences of the L-value expressions given by the kids are affected by the barrier. Examples of L-value expressions are **LDA**s, **ILDA**s, **LDID**s of pointers, and any address expressions.

Barriers never have any effect on variables that are not modifiable or visible in the source program. This includes: pregs, constants, read-only variables, the base address of formal parameters that are passed by reference, the index variable of any **DO_LOOP** that encloses the barrier. Barriers also have *no* effect on objects declared volatile. It is an error to specify the L-value of these objects as kids in named barriers. Barrier semantics also implies liveness: the store to an object should not be regarded as dead if it reaches a barrier that affects it. The reason is because another thread of the PU executing at the same time may reference the object.

In the case of unnamed barriers, to prevent the loss of too many optimization opportunities, private varables are excluded from being affected by the barrier. Variables are declared to be private (local) or shared via MP pragmas. An auto variable is never shared unless its symbol table entry is marked with the ST_IS_SHARED_AUTO flag.

- **BACKWARD_BARRIER** **[VH—VL]**
  This operator designates a barrier to the code movement of memory access instructions in the backward direction (against the flow of control), used for MP support. The memory accesses affected by the barrier are specified in the same way as **FORWARD_BARRIER**. See **FORWARD_BARRIER** regarding rules for determining the affected objects.

- **DEALLOCA** **[VH—VL]**
  This operator restores the stack pointer (**$sp**) back to the value represented by Kid 0. Kid 0 must be a pointer that gets its value via an earlier **ALLOCA** with size 0. Kids 1 and up are dummy operands that give pointers or address expressions to the allocated objects that are the left-hand-sides of the affected **ALLOCA**s, whose dereferences are no longer valid because their pointed-to memory areas have been deallocated by this operator. Kids 1 and up are to be regarded as L-value occurrences (i.e. stores) of the pointed-to locations by the compiler components, so that movements of their dereferences can be automatically blocked by this statement. A compiler-generated **ALLOCA** must lead to a **DEALLOCA** in which the pointer to the allocated block is specified as one of the dummy operands. For user-specified **ALLOCA**s, since the affected dereferences cannot be easily collected, a **DEALLOCA** with no dummy operand (i.e. only Kid 0) can be specified, which will block the movement of *all* dereferences. For user-specified **ALLOCA**s, **DEALLOCA** is generated only by the inliner: when the inliner inlines a procedure that contains a user-specified **ALLOCA**, it must insert an **ALLOCA** with 0 argument at the start of the inlined body, and a corresponding **DEALLOCA** with no dummy operand at each

each exit from the inlined body, to preserve the original stack allocation and deallocation behavior of the program, and prevent the movement of dereferences beyond the deallocation points.

## 1.9 Memory Accesses

In WHIRL, program variables and static data are regarded as being organized in blocks of memory. The blocks of memory can be allocated statically, or automatically on procedure entry in the procedure's stack frame. One important job of the compiler is to lay out the variables and data in memory so that the operations that access them can be performed by an efficient sequence of instructions.

Memory accesses in WHIRL are represented by load and store operations. These operations are either direct or indirect. The operators for direct load and store are **LDID** and **STID** respectively. They are used whenever the address of the accessed data is fixed.

Directly accessed locations are specified in WHIRL by the triple — *st_idx, offset* and *size*. Each symbol table entry has a field that specifies the *block*. Each separately declared object is assigned a unique block. The symbol table entry of the object has another offset field, which gives the offset of the object within the block. The real offset of an accessed location within the block is given by the sum of the offset in the WHIRL node and the offset in the symbol table entry. The size is implied by the descriptor type.

The purpose off the offset field in the symbol table entry is to enable memory layout to be performed by just updating the symbol table entries. As compilation progresses, the SGI Pro64 back-end components perform layout of the program variables by collescing them from a large number of smaller blocks into a small number of large blocks. As each variable is laid out in a block, its offset field in the symbol table entry is adjusted to reflect the new offset within the larger block. All compile-time data layout has to be completed before lowering to L WHIRL. In L WHIRL, the symbol table entry referenced in the WHIRL node must have 0 offset, so that the full offset within the block is given in the offset field in the WHIRL node.

**LDID** and **STID**, enable the compiler to do a better job in optimizing the memory accesses, due to the fact that the locations are known to the compiler, and the compiler knows that it is dealing with a specific data object. Having exact location information allows the compiler to more efficiently check for the presence of aliasing. Given two direct accesses, the compiler can verify that there is no alias among them by just checking that there is no overlap between the accessed locations. If the address of the location is never taken, the compiler can assume that any other indirect accesses will not affect the location. Having accurate alias information allows the optimizer more freedom in moving expressions that contain memory references around.

Indirect memory accesses are represented by **ILOAD** and **ISTORE** respectively. These operators reference an expression that computes the address of the location being accessed. It takes substantially more compilation time to do an accurate job of determining the possible locations that an **ILOAD** or **ISTORE** accesses. The work involves carrying around ranges of values and tracing the contents of pointers. After all the possible locations have been determined, it still has to find all the data objects that alias with these locations. When compilation speed is important, such expensive analyses have to be omitted, and the compiler has to assume the worst cases regarding aliases for the indirect loads and stores. As a result, direct memory accesses using **LDID** and **STID** are always preferred over indirect memory accesses, and the optimizer will try to promote an **ILOAD** or **ISTORE** to **LDID** or **STID** whenever it can determine that the computed address is a constant.

In WHIRL, stores are statements and loads are expressions. **LDID** is a leaf, because it does not use the result of any other computation. For all the load operators, **desc** specifies the data type in memory, while **res** specifies the data type in the hardware register. In VH WHIRL, the data types can be any type, but in M WHIRL and lower, type M is not allowed. For other than integer types, **res** and **desc** must be the same type. For integer types, **res** and **desc** must be the same type differing only by size. For the store operators, **desc** specifies the data type in memory, while **res** must be type V.

For fields within a struct or union, the additional annotation of field_id is provided. All the nested fields in a struct are flattened and a unique integer number is assigned to each field. This allows more accurate information to be represented in the case of overlapping fields. If a struct is itself a field within another struct, the struct itself is also given a field_id. The field_id of 0 is given to the top-level un-nested struct. All the nested structs and fields inside it are assigned integer numbers starting from 1. The ty_idx field in the WHIRL node must give the type of the outermost struct within which field_id's are assigned whenever field_id is not 0.

Since field_id uniquely identifies a field, the exact layout of the field within the struct can be delayed. Prior to this layout, the offset field in the WHIRL node is the offset for the top-level un-nested struct. In the current implementation, only the layout of bit-fields are delayed. For non-bit-fields, the field_id only provides supplemental information, and is not required for code generation purpose.

Since the field_id field is only 14 bits long, it is not large enough in the case of structs that have more than 16383 fields. As a result, we reserve the value 16383 to mean unknown or unrepresentable field, which also occurs when optimization generates an access that does not correspond to any particular field. If field_id cannot be used, then bit-field accesses cannot be represented in this form, and the lowered bit-field operators of **LDBITS**, **STBITS**, **ILDBITS** and **ISTBITS** must be used.

- **LDID** **res=B,i,f,z,M desc=all** **[VH—VL]**
  This operator contains st_idx, field_id and offset. This specifies a direct load from the address in bytes given by the offset field, located within the block given by the symbol table entry. This node also contains a ty_idx that gives the high level type of the object, which includes the volatile attribute. Type M is allowed only in VH and H WHIRL. Type B for **desc** is allowed only if the object is a register, and **res** can be type B only if **desc** is also type B.

- **STID** **desc=all** **[VH—VL]**
  This operator contains st_idx, field_id and offset. This specifies a direct store of the value computed by Kid 0 to the address in bytes given by the offset field, located within the block given by the symbol table entry. This node also contains a ty_idx that gives the high level type of the object, which includes the volatile attribute. Type M is allowed only in VH and H WHIRL. Type B for **desc** is allowed only if the object is a register.

- **ILOAD** **res=i,f,z,Mdesc=all** **[VH—VL]**
  A load or dereference is performed from the address in bytes given by adding the offset field to the address computed by Kid 0. This node contains two ty_idx's, one giving the high level type of the pointer through which the indirection is performed, and the other giving the high level type of the item being loaded. If the loaded object is a field in a struct, field_id identifies the exact field. Type M is allowed only in VH and H WHIRL.

- **ILOADX** **res=f** **desc=f** **[M—VL]**
  This operator is only generated by later phases of the compiler. This operator contains two kids. Both kids must be **LDID**s corresponding to two

pseudo-registers. This operator loads from the address given by the sum of the two pseudo-registers. Two ty_idx's are provided as in **ILOAD**.

- **MLOAD** **[M—L]**
  A multiple-byte load is performed from the address in bytes given by adding the offset field to the address computed by kid 0. Kid 1 gives the number of bytes to load.This node contains a ty_idx that gives the high level type of the pointer through which indirection is performed. If the loaded object is a field in a struct, field_id identifies the exact field.

- **ISTORE** **desc=all** **[VH—VL]**
  A store of the value computed by Kid 0 is performed to the address in bytes given by adding the offset field to the address computed by Kid 1. This node also contains one ty_idx that gives the high level type of the pointer through which indirection is performed. If the stored-to object is a field in a struct, field_id identifies the exact field.Type M is allowed only in VH and H WHIRL

- **ISTOREX** **desc=f** **[M—VL]**
  This operator is only generated by later phases of the compiler. This operator contains three kids. Kids 1 and 2 must be **LDID**s corresponding to two pseudo-registers. This operator stores the value computed by Kid 0 to the address given by the sum of the two pseudo registers. ty_idx is provided as in **ISTORE**.

- **MSTORE** **[M—L]**
  A multiple-byte store of the value computed by Kid 0 is performed to the address given by adding the offset field to the address computed by Kid 1. Kid 2 gives the number of bytes to store. This node also contains ty_idx that gives the high level type of the pointer through which indirection is performed. If the stored-to object is a field in a struct, field_id identifies the exact field. Kid 0 is either an **MLOAD** or a scalar expression. If Kid 0 is an **MLOAD**, it must be of the same size, and there must be no overlap between the source and target memory. If Kid 0 is a scalar expression, the size of the **MSTORE** must be a multiple of the size of the type of the scalar expression, and the alignment of the start address of the **MSTORE** must also match the alignment of the type of the scalar expression.

## 1.10 Bit-field Representation

Since bit-fields are always fields in a struct, they can be represented by field_id in the load and store WHIRL operations. The data type BS is used in **desc** to indicate bit-field loads and stores, in which case the offset field gives the offset of the top-level un-nested struct.

Bit-field loads and stores have to be lowered in getting to M WHIRL. The lowered forms of bit-field loads and stores are also used whenever field_id cannot be used, which could be due to bit-field optimizations or because the field number exceeds the size of the field_id field. In **LDBITS**, **STBITS**, **ILDBITS** and **ISTBITS**, field_id is replaced by a pair of numbers, bit_offset and bit_size, that give the offset and length respectively of the bit-field being accessed. In these operators, **desc** gives the unit of memory being accessed in order to extract or deposit the bit-field.

**EXTRACT_BITS** and **COMPOSE_BITS** are even lower-level operations related to bit-fields. They should be generated only if the target instruction set provides similar instructions.

- **LDBITS res=i desc=i,I1,I2 [VH—VL]**
  This operator corresponds to an **LDID** with field_id 0. **desc** gives the unit of memory being loaded before the bit-field extraction. The bit-field extraction is specified by the fields bit_offset and bit_size.

- **STBITS desc=i,I1,I2 [VH—VL]**
  This operator corresponds to an **STID** with field_id 0. **desc** gives the unit of memory being accessed to perform the bit-field deposition. The bit-field deposition is specified by the fields bit_offset and bit_size.

- **ILDBITS res=i desc=i,I1,I2 [VH—VL]**
  This operator corresponds to an **ILOAD** with field_id 0. **desc** gives the unit of memory being loaded before the bit-field extraction. The bit-field extraction is specified by the fields bit_offset and bit_size.

- **ISTBITS desc=i,I1,I2 [VH—VL]**
  This operator corresponds to an **ISTORE** with field_id 0. **desc** gives the unit of memory being accessed to perform the bit-field deposition. The bit-field deposition is specified by the fields bit_offset and bit_size.

## 1.11 Pseudo-registers

One important task of the compilation process is to identify candidates for allocation to registers. WHIRL programs can use an unlimited number of pseudo-registers. An important property of pseudo-registers is that they are never aliased to anything. This simplifies the job of the global register allocator (GRA) in CG, which will map all the pseudo-registers to the set of physical registers in the target machine. In this process, it may have to spill some of them back into memory, or re-materialize them to avoid the memory store operations.

Pseudo-registers (pregs) do not need to have symbol table entries, because they do not correspond to user variables, and do not need to be laid out in memory unless spilled. But because they resembles memory objects, we refer to them using **LDID**s and **STID**s. However, their addresses cannot be taken using **LDA**.

The symbol table entry given by the **LDID** or **STID** will identify the object as being a preg. The offset field in the WHIRL node gives the number of the preg being accessed. The preg number is unique within the entire PU, and their numbering starts from 1. Preg 0 is reserved and disallowed for use. All pregs of the same data type will point to the same symbol table entry. Pregs of all the WHIRL data types except V and M are allowed. For integer types, pregs must be either 32-bit or 64-bit, since the C language specifies that intermediate values of computation can only be of these two sizes; starting in M WHIRL, if **desc** gives a size smaller than the physical size of the register in the compilation target, it indicates that that the high-order bits of the register are not live.

Since pregs have to correspond to the hardware registers, starting in L WHIRL, only the data types that have exact correspondence to the hardware registers are allowed in pregs. Pregs for the complex data types are lowered to pairs of float pregs in M WHIRL. Pregs for quad-word floats are lowered to pairs of float pregs in L WHIRL. Pregs of type B are introduced starting in M WHIRL, and they correspond to predicate registers.

**LDID**s and **STID**s of pseudo-registers do not cause implicit type conversions to be generated. The same floating-point pseudo-register is not allowed to be F4 and F8 at different places in the same program unit. For integer data types, the same pseudo-registers may be referenced as I4, I8, U4 or U8 at different places because the compiler recognizes that some integer type conversions are no-op. Type conversions for pseudo-registers that are not no-op must be represented explicitly by conversion operators in WHIRL so that they can be optimized by the WHIRL optimizer.

Whenever the compiler needs to save the intermediate results of computations, it should generate and use new pregs whenever possible, as opposed to temporaries that reside in memory, because this avoids the overhead of creating and maintaining symbol table entries, and they do not have to be allocated in memory unless spilled. Subsequent compiler phases also have less overhead dealing with pregs because they are never aliased. In contrast, temporaries are regarded as memory objects, and symbol table entries have to be created for them.

As compilation proceeds in the back-end, pregs are generated to store intermediate results. In the register variable identification (RVI) phase, the compiler attempts to convert as many memory accesses to preg accesses as possible, while leaving behind the minimum number of memory loads and stores. This phase also attempts to allocate constants to registers. As a result, a data value can reside in pregs and memory at different places in the program.

We call pregs that have home memory locations *has-home* pregs. A home can be associated with only one preg, and a has-home preg can be associated with only one home. The live range of a preg is the set of WHIRL statements over which it is both defined and live. Over the live range of a has-home preg, its home location cannot be assumed to contain up-to-date values. The only exception is in the case where a has-home preg has only uses over a contiguous part of its live range, in which case the home location can be regarded as having valid content over that region.

Depending on the target machine, different classes and numbers of physical (or dedicated) registers can show up starting in M WHIRL. They are identified by different symbol table entries. Their usages are associated with the passing of function parameters and return values or compilation regions. In L WHIRL, additional dedicated pregs will be manifested that reference the global pointer, the frame pointer and the return address register.

Dedicated pregs are not subject to the fixed-size restriction as for ordinary pregs. Each floating-point preg can be both F4 and F8 at different times, if the target ISA allows. Dedicated pregs are not re-mapped in later code generation phases.

The special preg -1 is used in VH and H WHIRL for specifying the return value of a function call. Preg -1 can be used only once after each call that sets its value. In VH and H WHIRL, preg -1 suffices because a function can return only one item, even though it may be a composite item. After lowering to M WHIRL, depending on the linkage convention, more than one item can be returned in multiple dedicated registers. See Section 1.7 regarding restrictions on where negative pregs can appear.

## 1.12 Other Leaf Operators

Apart from **LDID**, these are the other operators that constitute leaves in WHIRL trees:

- **LDA** **res= A4,A8** **[VH—VL]**
  Return the address in bytes given by adding the offset field to the address of the symbol given by st_idx. The symbol can be either a variable or a function. This node also contains ty_idx that gives the high level type of the address being loaded.

- **LDMA** **res= A4,A8** **[VH—VL]**
  Same as **LDA**, but the address cannot be regarded as constant because it is mutable, in the sense that the address of the variable or function may be changed by a procedure call. There are two situations in which the address of a symbol can be changed by a procedure call. In the first situation, the call causes a new dynamic object to be linked in, and the definition of the symbol is pre-empted by it. (Dynamic objects can be linked in at run-time via the *dlopen(2)* or *sgidladd(2)* symtem calls). The second situation applies only to functions, and is due to lazy-text resolution performed by the run-time linker, or quickstart. For the second situation, the address of the function is changed only when it is called the first time. A symbol is mutable only if its export class is EXPORT_PREEMPTIBLE. In the case of variables, it must additionally be either a weak symbol or is of the SCLASS_COMMON or SCLASS_EXTERN storage class.

- **LDA_LABEL** **res= A4,A8** **[VH—VL]**
  Return the text address of the label_number given. This node also contains ty_idx that gives the high level type of the address being loaded, which should be a pointer to void.

- **IDNAME** **[VH—VL]**
  Refer to the name of a symbol given by st_idx and offset. This is used for the formal parameters in **FUNC_ENTRY** and **ALTENTRY**, and for the induction variable in **DO_LOOP**. This operator is not executable, and is for specification purpose only.

- **INTCONST** **res=B,i** **[VH—VL]**
  Return an integer value. The integer value is contained in the 64 bit field const_val. When representing a 32-bit integer, the high-order 32 bits are ignored.

- **CONST** **res=i,f,z** **[VH—VL]**
  Return a literal value. st_idx points to the entry that gives the literal value. For the integer types, this operator is used to specify symbolic constants.

## 1.13 Type Conversions

In this section, we talk about the type conversion operators **CVT** and **CVTL**, and the treat-as operator **TAS**. These operators have data types that are different between their operands and results. **CVT** and **CVTL** maintain the same value, while changing the representation from one type to another. **TAS** preserves the bit representation and interprete the value as if it is of a different type.

To effectively serve as the medium to perform optimizations for the underlying target machine, it is most ideal for one operation in WHIRL to map to exactly one machine instruction. If there are more operations in WHIRL than after they have been translated to machine instructions, any common subexpression that the optimizer recognizes at the WHIRL level could be wrongly disguised, causing unnecessary saving of the disguised common subexpression and the unnecessary occupation of a register. VH and H WHIRL are target-independent. Starting in M WHIRL, we discourage the generation of any WHIRL operation that translates to a no-op in the target machine.

**CVT** is used for conversions among the data types i and f. To support integer values represented by smaller number of bits, **CVTL** is used. The integer value must still be manipulated in register as one of the base types I4, I8, U4 or U8. In between operations, **CVTL** is used to effect truncation and sign extension within the base type.

The purposes of **CVT** and **CVTL** are to preserve the value while changing representation. For some conversions, the value being converted may be unrepresentable in the new representation because it lies outside the domain of the result type. The compiler always generates code that does the correct conversion for in-range values, and the correct truncation for out-of-range values. A special case occurs when a negative signed integer is converted to unsigned; in this case, the result is really undefined. However, consistent results can be produced by generating different code according to how the size changes: if the size is unchanged or increased, no code is generated, which means that the value is sign-extended; if the size decreases, the signed value is truncated.

**TAS** is always a no-op except when casting between floating-point and integer types. **TAS** takes a ty_idx that gives the high level type description of the casted result. In cases where the high level type information given is crucial for optimization purposes, the **TAS** should be generated even if it translates to a no-op. Any transformation done to the code around the **TAS** must not destroy the type information given by it. As a result, **TAS** is a barrier to tree restructuring transformation, similar to the **PAREN** operator.

- **CVT** **res=i,f** **desc=B,i,f** **[VH—VL]**
  The value in Kid 0 is converted from type **desc** into type **res**. For conversions from **f** to **i**, **CVT** can map to one of **RND**, **TRUNC**, **FLOOR** and **CEIL** depending on the rounding mode set in the target processor. In both Fortran and C, conversion from floating point to integer is defined to use the truncation semantics, so the front-ends should explicitly use **TRUNC** for such type conversions. Conversion from B to **i** corresponds to transferring the boolean value from a predicate register to an integer register.

- **CVTL** **res=i** **[VH—VL]**
  The value computed by Kid 0 is to be treated as being of the given size in number of bits represented by the basic type **res**. The type of Kid 0 must be of the same size as **res**. For **res**=U8 or U4, the rest of the bits are made to be zero. For **res** = I8 or I4, the rest of the bits are sign-filled. The size specified in the node must be smaller than the size of **res** in bits.

- **TAS** **res=i,f** **[VH—VL]**
  Treats (or casts) the value computed by Kid 0 as being of type **res**. The bit representation of the value is unchanged. The type of Kid 0 must be of the same size as **res**. A ty_idx is used to give the high level type description of the result type.

## 1.14 High Level Type Specification

High level types are the composite types that users specify in their programs. They provide additional type information beyond that provided by the data type fields in the WHIRL node. Since high level types have built-in structure and hierarchy, they can only be represented in the symbol table via the TY entries. There are ty_idx fields in the symbol table entries that give the declared type of each variable. But in modern programming languages, type information is not just limited to the places in the program where things are declared. Languages like C allow type casts in executable statements that can alter the semantics of the computation. As a result, ty_idx's are provided in a few WHIRL operators to carry the type casting information from the original program. High level type information in WHIRL serves the following purposes:

1. It provides the complete information to allow correct code generation: Information like *alignment* and the *volatile* attribute is carried in the high level type information in WHIRL.

2. It enables better optimizations: Under some options, (for example, "–TENV:alias=typed"), the compiler can assume that accesses to objects through pointers to different types are not aliased to each other. This allows the compiler to more aggressively move memory references around to achieve better performance.

3. It supports translation of WHIRL back to the source language: The tools whirl2c and whirl2f can more accurately reconstruct the original program using the high level type information.

Whenever the data type fields in the WHIRL node provide sufficient information for a given translation or optimization, use of the data type fields should be preferred over high level types.

Since explicit type casts do not arise frequently, setting up a ty_idx field for all operators would unnecessarily expand the WHIRL node. We have chosen to provide ty_idx only for a few operators, **LDID**, **STID**, **LDA**, **ILDA**, **ILOAD**, **MLOAD**, **ISTORE** and **MSTORE**. To represent type casts that are not associated with these operations, we use **TAS** to specify the high level type. We now describe the ty_idx's in these operators:

**LDA, ILDA**— ty_idx gives the high level type of the address being loaded. If the address is subsequently dereferenced, it is assumed that the pointed-to object is dereferenced, and that the operation can only affect the block of memory locations whose size is the size of the type pointed to by the pointer type specified by the ty_idx.

**LDID** and **STID** — ty_idx gives the type of the object being loaded or stored into.

**ILOAD** — There are two ty_idx's, one for the pointer as computed by the address expression and the other for the result of the load. The result type cannot be derived from the address type only in the case of explicit type casting for the result of the load.

**MLOAD** — There is only one ty_idx that gives the type of the pointer computed by the address expression. The type for the object being loaded is not specified, as it can be inferred from the type of the address, and type casts to structs are not allowed in the languages supported.

**ISTORE** and **MSTORE** — Only the ty_idx for the pointer computed by the address expression is provided. The type of the value being stored can be determined by looking at the expression that computes the value.

**TAS** — This operator arises only from implicit or explicit type casts in the original program. The ty_idx gives the casted-to type. If the ty_idx can be carried with one of the above operators, this operator should not be generated.

In recognizing common subexpressions, the WHIRL optimizer (WOPT) handles the ty_idx in **TAS** differently from the other operators. Ordinarily, the optimizer disregards ty_idx's in recognizing common subexpressions. This is possible because the values computed by the two instances are the same, even if their ty_idx's are different. For example, if two loads are common subexpressions, they must be loading the same value from the same address. The process of recognizing common subexpressions will result in the optimizer using only one node to represent the two instances; the optimizer just randomly picks one of the ty_idx's to use in the single node. We do not think this will cause any error in the generated code, even if the compilation is "−TENV:alias =typed". On the other hand, this allows more common subexpressions to be recognized.

For **TAS**'s, WOPT includes the ty_idx in recognizing common subexpression. This means that two **TAS**'s with different ty_idx's will not be recognized as common subexpressions. This guarantees that optimization will never delete any high level type information provided in **TAS**'s.

The reason that we provide the address ty_idx in **ILOAD** and **ISTORE** is because the address expression referenced by them may not provide a result type ty_idx. For example, if the root of the address expression is an **ADD**, there is no ty_idx that gives the high level type of the result of the address expression. Such high level type information is needed in code generation and optimization for **ILOAD** and **ISTORE**.

The use of **TAS** that does not map to any machine instruction can cause non-optimal code sequences to be generated. This is illustrated in Figure 3. The occurrences of **TAS**'s cause the optimizer to use two registers instead of one in order to handle the common subexpression in **TAS**'s.

Input code:

```
   U4U4LDID p
  U4TAS t1
     :
   U4U4LDID p
  U4TAS t1
     :
  U4U4LDID p
     :
```

Optimized code:

```
  U4U4LDID p
 U4U4STID preg1
    U4U4LDID preg1
  U4TAS t1
 U4U4STID preg2
  U4U4LDID preg2
     :
  U4U4LDID preg2
     :
  U4U4LDID preg1
     :
```

**Figure 3 Effects of CSEs on TAS's**

C expression: *(((t1 *) (p+5)) + 4)

WHIRL expression:

```
     U4U4LDID p
     I4INTCONST 20
   U4ADD
  U4TAS t1
  I4INTCONST 16
 U4ADD
I4I4LOAD 0
```

**Figure 4 Example of appearance of TAS**

The example in Figure 3 shows that **TAS**'s should not be generated whenever possible. With our specification, a **TAS** would not have been generated if it is underneath a **ILOAD**, or associated with an **LDID** or **LDA**. So the situation where it has to appear should be very rare. Figure 4 gives an example of a situation where **TAS** has to be generated.

## 1.15 Expression Operators

In this section, we specify the WHIRL operators that are internal nodes in expression trees. We classify them according to the number of operands involved in the operation.All floating-point arithmetic operations, where applicable, are all intended to have the standard IEEE 754 semantics, including traps according to the current machine state.

### 1.15.1 Unary Operations

- **NEG** **res=i,f,z** **[VH—VL]**
  Return the arithmetic negation of Kid 0.

- **ABS** **res=i,f** **[VH—VL]**
  Return the absolute value of Kid 0.

- **SQRT** **res=f** **[VH—VL]**
  Return the sqrt of Kid 0.

- **RSQRT** **res=f** **[VH—VL]**
  Return the recipricol sqrt of Kid 0.

- **RECIP** **res=f** **[VH—VL]**
  Return the reciprical of Kid 0.

- **FIRSTPART** **res=f** **desc=FQ,z** **[VH—M]**
  For **res**=z, it returns the real part of the complex number given by Kid 0. For **res**=FQ, it returns the high part of the FQ value given by Kid 0. **res**=z is supported only in VH and H WHIRL. **res**=FQ is supported only in M WHIRL.

- **SECONDPART** **res=f** **desc=FQ,z** **[VH—M]**
  For **res**=z, it returns the imaginary part of the complex number given by Kid 0. For **res**=FQ, it returns the low part of the FQ value given by Kid 0. **res**=z is supported only in VH and H WHIRL. **res**=FQ is supported only in M WHIRL.

- **PAREN** **res=f,z** **[VH—VL]**
  Place a parenthesis around the expression in Kid 0. This is used to force the order of evaluation on an expression.

- **RND** **res=i** **desc=f** **[VH—VL]**
  Return Kid 0 rounded to the nearest integer.

- **TRUNC** **res=i** **desc=f** **[VH—VL]**
  Return Kid 0 rounded towards zero.

- **CEIL** **res=i** **desc=f** **[VH—VL]**
  Return Kid 0 rounded towards $+\infty$.

- **FLOOR** **res=i** **desc=f** **[VH—VL]**
  Return Kid 0 rounded towards -∞.

- **BNOT** **res=i** **[VH—VL]**
  Return the bitwise not of Kid 0.

- **LNOT** **res=B,i** **desc=B,i** **[VH—VL]**
  Return the logical not of Kid 0. The operand and result must both be of type boolean.

- **LOWPART** **res=i** **[M—VL]**
  Operate on an **LDID** of a preg that contains the result of an **XMPY** or **DIVREM** and return the part that represents the low-order part of the multiply or quotient of the divide respectively.

- **HIGHPART** **res=i** **[M—VL]**
  Operate on an **LDID** of a preg that contains the result of an **XMPY** or **DIVREM** and return the part that represents the high-order part of the multiply or remainder of the divide respectively.

- **MINPART** **res=i** **[M—VL]**
  Operate on an **LDID** of a preg that contains the result of an **MINMAX** and return the part that represents the minimum.

- **MAXPART** **res=i** **[M—VL]**
  Operate on an **LDID** of a preg that contains the result of a **MINMAX** and return the part that represents the maximum.

- **ILDA** **res= A4,A8** **[VH]**
  Return the address in bytes given by adding the offset field to Kid 0 . The symbol can be either a variable or a function. This node also contains ty_idx that gives the high level type of the pointer corresponding to Kid 0. If the address being loaded corresponds to a field in a struct, field_id identifies the exact field. This operator can be viewed as computing the l-value of an **ILOAD** that has the same contents and kid.

- **EXTRACT_BITS** **res= I4,I8,U4,U8** **[VH—VL]**
  Perform a bit-field extraction, specified by the fields bit_offset and bit_size, on the value computed by Kid 0. The value of the extracted bit-field is returned. This instruction is more general than **LDBITS**/**ILDBITS**, and may be generated as a result of lowering them.

- **PARM** **res=i,f,z,M,V** **[VH—VL]**
  This must be a kid of **CALL**, **ICALL**, **VFCALL**, **PICCALL**, **INTRINSIC_CALL** or **INTRINSIC_OP**. It specifies that Kid 0 is an actual parameter in the call. **res** is allowed to be V only in VH WHIRL, in which case it has no kid. ty_idx gives the high level type of the parameter (as given by the function prototype). The flags field gives different attributes about the parameter: *call-by-reference*, *in (call-by-value)* and *out*. The *dummy* attribute specifies that the parameter is present only to carry the right alias information to the optimizer, and code to pass the parameter does not need to be generated. There are additional attributes to represent the results of alias analysis: *read-only* indicates that the reference parameter being passed is referenced but not modified; *passed-not-saved* indicates that the callee does not save the address passed; *not-exposed-use* indicates that there is no exposed use of the passed value in the callee; *is-killed* indicates that the reference parameter is definitely assigned to in the callee.

- **ASM_INPUT** **res=i,f,z** **[VH—VL]**
  This must be a kid of **ASM_STMT**, and specifies that Kid 0 is an expression whose value is the input operand. The st_idx field gives a CLASS_NAME symbol table entry whose name is the operand's constraint string.

- **ALLOCA** **res=A4,A8** **[VH—VL]**
  Return a pointer to the block of uninitialized local stack space allocated by adjusting the stack pointer. Kid 0 gives the size in bytes of the block of memory to be allocated. This operator must only appear as the right-hand-side of a store statement. A zero value for the operand can be used to get the current base of the stack frame without any allocation. There are two kinds of **ALLOCA**s: user-specified and compiler-generated. See **DEALLOCA** for additional usage requirements for this operator.

### 1.15.2 Binary Operations

- **PAIR** **res=FQ,z** **[VH—M]**
  For **res**=z, it creates a complex number whose real part is equal to the value in Kid 0 and whose imaginary part is equal to the value in Kid 1. For **res**=FQ, it creates a FQ number from the high part given by Kid 0 and the low part given by Kid 1. **res**=z is supported only in VH and H WHIRL. **res**=FQ is supported only in M WHIRL.

- **ADD** **res=i,f,z** **[VH—VL]**
  Return Kid 0 plus Kid 1.

- **SUB** **res=i,f,z** **[VH—VL]**
  Return Kid 0 minus Kid 1.

- **MPY** **res=i,f,z** **[VH—VL]**
  Return the result when Kid 0 is multiplied by Kid 1. In M WHIRL or lower, for type integer, this operator can alternatively be represented by **XMPY** followed by **LOWPART** so that the multiply operation can be commonized with respect to another **HIGHMPY** of the same operands.

- **HIGHMPY** **res=i** **[VH—VL]**
  Return the high-order part of the result when Kid 0 is multiplied by Kid 1. In M WHIRL or lower, this operator can alternatively be represented by **XMPY** followed by **HIGHPART** so that the multiply operation can be commonized with respect to another **MPY** of the same operands.

- **XMPY** **res=i** **[M—VL]**
  Return the composite result when Kid 0 is multiplied by Kid 1. This operator is lowered from either **MPY** or **HIGHMPY**, and its result can only be operated on by **LOWPART** and **HIGHPART**. Though its result is actually made up of a pair of values, it can be regarded as being of the same type at the WHIRL level. The code generator will deal with the details of handling the pair of values. After optimization, **XMPY** can only appear as a kid of an **STID** to a preg. The preg containing the result can only appear as the operand of **LOWPART** or **HIGHPART**.

- **DIV** **res=i,f,z** **[VH—VL]**
  Return the quotient when Kid 0 is divided by Kid 1. In M WHIRL or lower, for type integer, this operator can alternatively be represented by **DIVREM** followed by **LOWPART** so that the divide operation can be commonized with respect to another **REM** of the same operands.

- **MOD** **res=i** **[VH—VL]**
  Return Kid 0 modulus Kid 1. The modulus operator of the form *(i mod j)* is defined as the value of the expression $(i - k \times j)$ for some integer *k* such that the value of the expression falls in the range between 0 and *j* or is 0. The sign is the sign of the divisor. *–(–i mod –j)* yields the same value as *(i mod j)*. When the sign of the two operands are the same, it yields the same value as **REM**. When only one operand is negative and the result is not 0, *(i mod j) = (i % j) + j*.

- **REM** **res=i** **[VH—VL]**
  Return the remainder when Kid 0 is divided by Kid 1. This implements the % operation in C. *(a % b)* is defined as the value of the expression

$a - \frac{a}{b} \times b$. The sign is the sign of the dividend. *–(–a % –b)* yields the same value as *(a % b)*. When the sign of the two operands are the same, it yields the same value as **MOD**. In M WHIRL or lower, this operator can alternatively be represented by **DIVREM** followed by **HIGHPART** so that the divide operation can be commonized with respect to another **DIV** of the same operands.

**Table 3 Examples to show relationship between MOD and REM**

| a | b | a mod b | a rem b |
|---|---|---|---|
| 8 | 5 | 3 | 3 |
| -8 | 5 | 2 | -3 |
| 8 | -5 | -2 | 3 |
| -8 | -5 | -3 | -3 |

- **DIVREM** **res=i** **[M—VL]**
  Return the composite result representing both the quotient and the remainder when Kid 0 is divided by Kid 1. This operator is lowered from either **DIV** or **REM**, and its result can only be operated on by **LOWPART** and **HIGHPART**. Though its result is actually made up of a pair of values, it can be regarded as being of the same type at the WHIRL level. The code generator will deal with the details of handling the pair of values. After optimization, **DIVREM** can only appear as a kid of an **STID** to a preg. The preg containing the result can only appear as the operand of **LOWPART** or **HIGHPART**.

- **MAX** **res=i,f** **[VH—VL]**
  Return the maximum of Kid 0 and Kid 1.

- **MIN** **res=i,f** **[VH—VL]**
  Return the minimum of Kid 0 and Kid 1.

- **MINMAX** **res=i,f** **[M—VL]**
  Return the composite result representing both the minimum and the maximum when Kid 0 is compared with Kid 1. This operator is lowered from either **MAX** or **MIN**, and its result can only be operated on by **MAXPART** and **MINPART**. Though its result is actually made up of a pair of values, it can be regarded as being of the same type at the WHIRL level. The code generator will deal with the details of handling the pair of values. After optimization, **MINMAX** can only appear as a kid of an **STID** to a preg. The

preg containing the result can only appear as the operand of **MAXPART** or **MINPART**.

- **EQ** **res=B,i** **desc=B,i,f,z** **[VH—VL]**
  Return true if Kid 0 is equal to Kid 1, false otherwise.
- **NE** **res=B,i** **desc=B,i,f,z** **[VH—VL]**
  Return true if Kid 0 is not equal to Kid 1, false otherwise.
- **GE** **res=B,i** **desc=i,f** **[VH—VL]**
  Return true if Kid 0 is greater than or equal to Kid 1, false otherwise.
- **GT** **res=B,i** **desc=i,f** **[VH—VL]**
  Return true if Kid 0 is greater than Kid 1, false otherwise.
- **LE** **res=B,i** **desc=i,f** **[VH—VL]**
  Return true if Kid 0 is less than or equal to Kid 1, false otherwise.
- **LT** **res=B,i** **desc=i,f** **[VH—VL]**
  Return true if Kid 0 is less than Kid 1, false otherwise.
- **BAND** **res=i** **[VH—VL]**
  Return the bitwise AND of Kid 0 and Kid 1.
- **BIOR** **res=i** **[VH—VL]**
  Return the bitwise OR of Kid 0 and Kid 1.
- **BNOR** **res=i** **[VH—VL]**
  Return the bitwise NOR of Kid 0 and Kid 1.
- **BXOR** **res=i** **[VH—VL]**
  Return the bitwise XOR of Kid 0 and Kid 1.
- **LAND** **res=i** **[VH—VL]**
  Return the logical AND of Kid 0 and Kid 1. The children and the result are of type boolean. The code generated may use short-circuiting.
- **LIOR** **res=i** **[VH—VL]**
  Return the logical OR of Kid 0 and Kid 1. The children and the result are of type boolean. The code generated may use short-circuiting.
- **CAND** **res=i** **[VH—H]**
  Control flow version of LAND. It evaluates the logical AND of Kid 0 and Kid 1 via short-circuiting. Kid 1 is not to be evaluated if Kid 0 evalutes to

0. In VH WHIRL, the kids can contain side-effect operations (via **COMMA** and **RCOMMA**). If there are side effects, the lowered form in H WHIRL will use jumps.

- **CIOR** **res=i** **[VH—H]**
  Control flow version of LIOR. It evaluates the logical OR of Kid 0 and Kid 1 via short-circuiting. Kid 1 is not to be evaluated if Kid 0 evalutes to 1. In VH WHIRL, the kids can contain side-effect operations (via **COMMA** and **RCOMMA**). If there are side effects, the lowered form in H WHIRL will use jumps.

- **SHL** **res=i** **[VH—VL]**
  Return Kid 0 shifted left Kid 1 times. All the low order bits shifted in are set to zero. The exact semantics depends on the target architecture.

- **ASHR** **res=i** **[VH—VL]**
  Return Kid 0 arithmetically shifted right Kid 1 times. The exact semantics depends on the target architecture.

- **LSHR** **res=i** **[VH—VL]**
  Return Kid 0 logically shifted right Kid 1 times. The exact semantics depends on the target architecture.

- **COMPOSE_BITS** **res= I4,I8,U4,U8** **[VH—VL]**
  Creates a new integer value by performing bits composition using two operands. The value of Kid 1 is deposited into the range of bits in Kod 0 as specified by the fields bit_offset and bit_size. If the value of Kid 1 is larger than what the bit-field can contain, its value is truncated. The rest of the bits are taken from the value in Kod 0. The resulting new integer value is returned. **res** must be the same as that of Kid 0. This instruction is more general than **STBITS**/**ISTBITS**, and may be generated as a result of lowering them.

- **RROTATE** **res=U4,U8desc=U1,U2,U4,U8** **[VH]**
  Return Kid 0 rotated to the right by the number of bits specified by Kid 1. Only the low order part of Kid 0 corresponding to **desc** is used. The rotation amount must not be negative. Only the least significant bits of Kid 1 sufficient to specify the full bits in **desc** are used to determine the rotate amount; the higher order bits of Kid 1 are ignored. The high order bits of the result that lie outside of **desc** have undefined values.

- **COMMA** **res=i,f,z,M** **[VH]**
  Kid 0 must be a **BLOCK**, while Kid 1 must be an expression of type **res**. Kid 1 must not be another **COMMA**. The statements in the block given by

Kid 0 are executed before evaluating and returning the value of Kid 1. A call can be generated in the middle of an expression in VH WHIRL using this operator. If the return value of the call is to be used in the expression, Kid 1 can load the dedicated pseudo-register that contains the function return value.

- **RCOMMA** **res=i,f,z,M** **[VH]**
  Kid 0 must be an expression of type res, while Kid 1 must be a **BLOCK**. Kid 0 must not be another **RCOMMA**. The statements in the block given by Kid 1 are executed after evaluating Kid 1. The value of Kid 0 is returned.

## 1.15.3 Ternary Operations

- **SELECT** **res=i,f** **desc=B,i** **[H—VL]**
  Kid 0 must evaluate to a boolean expression. Both Kid 1 and Kid 2 must have **res** as the result type. Return Kid 1 if Kid 0 evaluates to true. Otherwise, return Kid 2. The evaluation of both Kids 1 and 2 can be performed regardless of the value of Kid 0. Converting an if statement to this operator is tantamount to speculation if Kid 1 or 2 are expressions.

- **CSELECT** **res=i,f,M,Vdesc=i** **[VH]**
  Control flow version of **SELECT**. The kids are the same as **SELECT**, but only one of Kid 1 and Kid 2 is to be evaluated depending on the result of Kid 0.

- **MADD** **res=f** **[VL]**
  Return (Kid 1 $\times$ Kid 2) + Kid 0.

- **MSUB** **res=f** **[VL]**
  Return (Kid 1 $\times$ Kid 2) – Kid 0.

- **NMADD** **res=f** **[VL]**
  Return – ((Kid 1 $\times$ Kid 2) + Kid 0).

- **NMSUB** **res=f** **[VL]**
  Return – ((Kid 1 $\times$ Kid 2) – Kid 0).

## 1.15.4 N-ary Operations

- **ARRAY** **res=A4,A8** **[VH—H]**
  This operator uses array addressing rules to return an address. The number of dimensions of the array, n, is inferred from kid-count shifted right

by 1. An internal field, element_size, gives the size of each array element in bytes. If element_size is negative, it specifies a non-contiguous array in FORTRAN90. Kid 0 is the address of the base of the array. Kids 1 to n give the size of each dimension in contiguous arrays, and the multiplier for each index in non-contiguous arrays. Kids n+1 to 2n give the index expressions for dimensions 0 to n-1 respectively (adjusted so that the array index has a zero lower bound). If we name Kids 1 to n as $m_1..m_n$, and if we name the values of the index expressions $x_1..x_n$ (i.e. $x_i$ = the value of Kid i+n), and if element_size is s, then for contiguous arrays, the resultant address is:

$$kid0 + s\sum_{i=1}^{n}\left(x_i \prod_{j=i+1}^{n} m_j\right)$$

and for non-contiguous arrays, the resultant address is:

$$kid0 + (-s)\sum_{i=1}^{n} x_i m_j$$

In contiguous arrays, for dimensions d=2..n, $0 <= x_d < m_d$; in other words, excepting the first dimension, each index expression must be in bounds.

- **INTRINSIC_OP res=I1,I2,U1,U2, i,f,z,M [VH—M]**
  This operator applies the intrinsic operation as specified by the intrinsic field to the operands specified by Kids 0..n-1, which must be **PARM** nodes, and returns the result. A flags field gives attributes about the intrinsic that are useful for optimization around the intrinsic. This operator can only be used for intrinsics that have no side effects and are pure functions. This means the value returned is dependent only on the arguments, which may be passed by reference. Depending on the intrinsic, its result type and compilation options, it will either become a call or a sequence of instructions after it is lowered to L WHIRL. The types I1, I2, U1, U2, M are only allowed in VH WHIRL.

- **IO_ITEM [VH—H]**
  This can appear only as kids of **IO**, and represents an item specified in a FORTRAN I/O statement. The intrinsic field gives the type of I/O item specified. This operator has either 0, 1, 2 or 3 kids depending on the type of I/O item. The kids are expression trees representing the contents of the I/O item. Call and **GOTO** statements are allowed to be nested within the expression tree. Thus, this operator can indicate implicit control flow.

## 1.16 Intrinsics

An intrinsic in WHIRL is an operation that cannot be mapped to exactly one machine instruction in the target architecture. However, there are some common language constructs that we exempt from this rule because they have common occurrences, like **CVTL**, **MAX** and **MIN**.

The list of intrinsics that WHIRL support is defined and maintained separately from the WHIRL operators. Both the call and the intrinsics operators carry attributes in the flags field that provide information to the compiler about the call or intrinsic operation. But intrinsics are distinct from calls because they represent “functions” that the compiler has special knowledge about and can take advantage of.

We support two intrinsic operators. **INTRINSIC_OP** is an expression operator, while **INTRINSIC_CALL** is a statement. The expression form allows the optimizer to treat the intrinsic the same as any other expression operator, so the intrinsic can benefit from any optimizations involving expressions, like common subexpression elimination. But because **INTRINSIC_OP** can only be defined for intrinsics that have no side effect, only a limit number of intrinsics can be represented under **INTRINSIC_OP**.

## 1.17 Aggregates Specification

Fortran 90 provides program constructs that represent aggregates of array elements in a compact form. Translation of such aggregate operations requires the introduction of loops. Operations on aggregates provide optimization opportunities that could be obscured or made more difficult once those operations are lowered into loops operating on array elements. Thus, we define VH WHIRL as the level of WHIRL that corresponds to program constructs as they appear in Fortran 90 programs. VH WHIRL constructs are also generated by Fortran 77 programs that use the 8X extensions.

In VH WHIRL, we allow a WHIRL node to specify an aggregate of values (as opposed to a single value). All WHIRL operators can take on aggregate values as operands. The **ARRAYEXP** operator is used to give the dimension information of an array expression.

Among the WHIRL operators for aggregates specification, **TRIPLET**, **ARRAYEXP** and **ARRSECTION** are expression operators. **WHERE** is a structured control flow statement.

- **TRIPLET** **res=i** **[VH]**
  This operator produces a one dimensional array of integers in a linear progression. Kid 0 evaluates to the starting integer value of the progression. Kid 1 evaluates to an integer value that gives the stride in the progression. Kid 2 evaluates to the number of values in the progression.

- **ARRAYEXP** **res=i,f,z** **[VH]**
  This operator indicates that Kid 0 is an array expression with the number of dimensions num_dim equal to the kid_count-1. Kid 1 to Kid num_dim give the number of elements for each dimension. An **ARRAYEXP** is required at the root of a tree that specifies array expressions. This means that it will occur at the statement level for aggregate stores. Within the tree, **ARRAYEXP** is not required unless an operand is of different shape (i.e. smaller number of dimensions) than what is expected by its parent. The **ARRAYEXP** node can also be used with only one child to indicate that the child expression is an array expression. This can occur due to the requirement that all array valued children of the **ARRSECTION** node are so indicated.

- **ARRSECTION** **res=A4,A8** **[VH]**
  This node corresponds to the **ARRAY**, except that it generates an aggregate of addresses. The number of indices is given by (kid_count-1)/2. The field element_size gives the size of each array element in bytes. Kid 0 is the address of the base of the array. Kids 1 to n give the sizes of all the dimensions of the array as declared. Each of Kids n+1 to 2n is either an integer expression or a one-dimensional array integer expression that indexes into the array at the corresponding dimension, adjusted so that the array index has a zero lower bound. The resulting array expression has a number of dimension corresponding to the number of kids from n+1 to 2n that are array expressions. It is required that each array-valued index child be either an **TRIPLET** or an **ARRAYEXP** of only one dimension, although the **ARRAYEXP** may be the marker (1 child) form.

- **WHERE** **[VH]**
  This is a structured control flow statement that implements the Fortran 90 masked assignment. It has three kids. Kid 0 must be a boolean-typed array expression that forms the mask. Kid 1 and 2 are **BLOCK**s consisting of only **ISTORE** nodes for aggregates of array elements. The shape of arrays or array sections being stored into must be the same as the shape of the boolean array expression of Kid 0. For each array element, either Kid 1 or Kid 2 is executed depending on the value of the mask. When an element of the mask in Kid 0 is true, only the stores specified in Kid 1 are performed to the corresponding elements of the arrays or array sections. When an element of the mask in Kid 0 is false, only the stores specified

in Kid 2 are performed to the corresponding elements of the arrays or array sections.

## 1.18 ASCII WHIRL Format

Although the WHIRL exists internally in the form of trees, it can be translated to the ASCII format for perusal. The IR portion of WHIRL has a standard ascii format that allows it to be edited and translated back to binary form. The symbol table portion of WHIRL, however, cannot be translated back to binary form. Thus, to produce a valid WHIRL binary file from ascii WHIRL, it is necessary to specify the original WHIRL file that contains the valid symbol table. When the ascii IR is translated back into binary form, the original symbol table is incorporated into the output WHIRL file.

In the ASCII WHIRL format, each line corresponds to one WHIRL node, with the name of the operator being the first field of each line. Additional fields in the node are displayed following the operator name. **res** and **desc** are printed as first and second prefixes of the operator name. By convention, the **res** or **desc** is omitted if there is only one legal type for that field allowed for that operator. For operators in which **desc** is always the same as, or can be derived from **res**, **desc** is also omitted.

In order to display the tree structures, ASCII WHIRL is a mixture of prefix and postfix notations. It takes advantage of the two-dimensional nature of the print-out by using indentations to display the nesting relationships. For each level down the tree, it indents by one more column to the right. This allows the kids of an operator to be identified easily by matching indentations. The indentation is for displaying purpose only. The ASCII-to-binary translator ignores indentations when it scans the content of each line.

Statements belonging to the same **BLOCK** are printed in the order of execution. Expressions are printed in postfix notation, while the structured control flow constructs are printed in prefix notation. This ensures that the order of appearances of the operands in WHIRL corresponds more closely to the generated assembler output.

To facilitate visual inspection and parsing by the ASCII WHIRL reader, keywords are inserted. Figure 4 shows the keywords used in displaying the structured control flow statements. The comment character # is used to specify that the rest of the line is to be ignored. This allows the compiler to insert information in the ascii WHIRL dump that helps debugging.

```
DO_LOOP                                 FUNC_ENTRY
  <index var>                             IDNAME
INIT                                      IDNAME
  <initialization statement>              . . .
COMP                                    BODY
  <comparison for end condition>          BLOCK
INCR                                      . . .
  <increment statement>                   END_BLOCK
BODY
  BLOCK
  . . .
  END_BLOCK                             DO_WHILE
                                          <index var>
                                        BODY
IF                                        BLOCK
  <condition>                             . . .
THEN                                      END_BLOCK
  BLOCK
  . . .
  END_BLOCK                             WHILE_DO
ELSE                                      <index var>
  BLOCK                                 BODY
  . . .                                   BLOCK
  END_BLOCK                               . . .
END_IF                                    END_BLOCK
```

(Inserted keywords are in bold face.)

**Figure 5 ASCII Formats for Structured Control Flow Statements**

In particular, the original text of the source line can be printed next to the WHIRL code generated from it.

# Index of Operators