Open64 Compiler
Whirl Intermediate Representation

August 3, 2007

Contents

Chapter 1

Whirl Abstract Syntax Tree

1.1 Introduction

This document discusses WHIRL, the intermediate language (IR) for the Open64 compiler. Using a common IR enables a compiler to support multiple languages and multiple processor targets. The different front-ends of the Open64 compiler translate the different languages to WHIRL. The Open64 compiler has a sophisticated 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 ofthe 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.

The WHIRL symbol table is described in Chapter 2

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 before 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

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

Table 1.1: Differences between high-level and low-level representations.

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 its level. Table 1.1 contrasts the general characteristics between high-level and low-level representations.

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 Open64 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 Open64 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 into its own internal representation that matches the target machine instructions. WHIRL thus serves as the common IR interface among all back end components. Because lowering is done only gradually, a secondary benefit is that each lowering step is simpler and easier.

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.

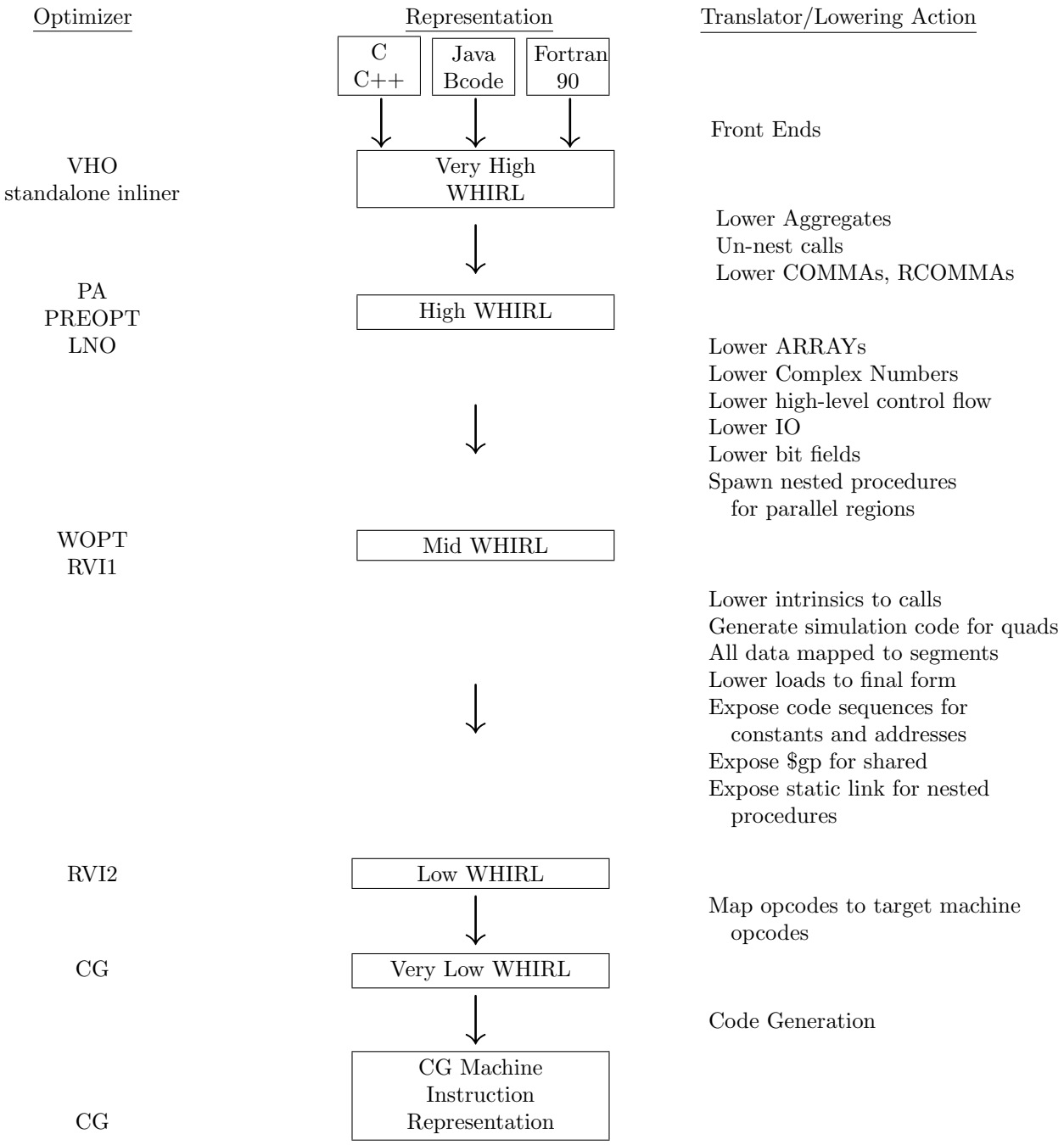


Figure 1.1: Continuous Lowering in the Open64 Compiler

The Very High WHIRL Optimizer (VHO) and stand-alone 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 LDAs, 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 document 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)
- I 1 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 BLOCKs. 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 1.2 gives the layout of struct WN.

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–VLWHIRL. 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 PRAGMAs that are relevant to the compilation of the PU. Kid n-2 is a BLOCK containing a list of PRAGMAs 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 callsites. Kid n-1 is a BLOCK node giving the body of the procedure.

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; no. of entries, COMPGOTO and SWITCH; length in bits for CVTL; label number; flags for calls, PARM and REGION; break code for TRAP, ASSERT;	word
byte 0	trip_est	estimated trip count for LOOP_INFO;	half-word byte
byte 2	depth	loop nesting depth for LOOP_INFO;	half-word
byte 4	st_idx	symbol table index; type index for all except LDA, LDID, STID; last label for COMPGOTO and SWITCH; number of exits for REGION; id for intrinsics flags field for PREFETCH, LOOP_INFO; 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; field ID for operators with fixed no. of kids; bit_offset at most significant 7 bits and bit_size at least significant 7 bits for LDBITS, STBITS, ILDBITS, ISTBITS, 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
byte 16	kids[0]	kid 0; first pointer for BLOCK; flags for LABEL;	word
byte 20	kids[1]	kid 1; type index for LDA, LDID, STID; address type pointer for ILOAD; 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 $n \geq 0$	word

Table 1.2: Layout of a WHIRL node

- 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 numberof 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 BLOCKs. 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_EXITs. The number of REGION_EXITs must be equal to the number of exits. Kid 1 gives a list of PRAGMAs 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 do-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 nonzero. 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 BLOCKs. Kid 1 gives 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/assignedgoto, 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 a 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 callnode 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 resultin 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 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 tothe 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 pregare 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 1.2.

- 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 tothe 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]

```
 PARM
  ..
  ..
 PARM
    LDID <field\_id for vptr>
   ILOAD <field\_id>
     ..
     ..
  ARRAY
 ILOAD <field\_id>
VFCALL
```

Figure 1.2: Form for VFCALL

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 PRAGMAs and/or XPRAGMAs 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.

- USE [VH–VL]

  This operator represents a Fortran USE statement. st_idx gives the module name. If the USE statement has an ONLY predicate the rType is set to MTYPE_B, otherwise it is MTYPE_V. If they are kids, they come in pairs. A pair signifies a potential rename. If a pair contains two identical nodes there is no actual rename, just a reference to a particular module symbol in conjunction with the ONLY predicate.

- 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 LDIDs 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 LDAs, ILDAs, LDIDs 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 variables 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 a respecified 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 ALLOCAs, 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) ofthe 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 ALLOCAs, 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 ALLOCAs, 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 Open64 back-end components

perform layout of the program variables by coalescing 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 efficient 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 preeferred 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, ecause it does not use the result of any other computation. For all of 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 nested structs and fields inside it are assigned integer numbers starting from 1. the ty_index 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-bitfields, 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,M desc=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 (use WN_load_addr_ty), and the other giving the high level type of the item being loaded (i.e., the referenced object; use WN_ty). 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 LDIDs 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 LDIDs 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 contain sty_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 bitfield 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 unitof 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 LDIDs and STIDs. 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 tableentry. 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 LWHIRL, 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 pre

LDIDs and STIDs 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 1.11 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 preempted by it. (Dynamic objects can be linked in at run-time via the dlopen(2) or sgidladd(2) system 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 fieldconst_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 interpret 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 cases 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.

```
Input code:

\index{LDID}%
 U4U4LDID p
U4TAS t1
  .
  .
\index{LDID}%
 U4U4LDID p
U4TAS t1
  .
  .
\index{LDID}%
U4U4LDID p
 .
 .
```

```
Optimized code:

 U4U4LDID p
\index{STID}%
U4U4STID preg1
  U4U4LDID preg1
 U4TAS t1
U4U4STID preg2
 U4U4LDID preg2
   .
   .
 U4U4LDID preg2
   .
   .
 U4U4LDID preg1
```

Figure 1.3: Effects of CSEs on TAS's

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 1.3. 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.

The example in Figure 1.3 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 1.4 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.

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

WHIRL expression:

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

Figure 1.4: Example of appearance of TAS

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 reciprocal sqrt of Kid 0.

- RECIP res=f [VH–VL]

  Return the reciprocal 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 MWHIRL.

- 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=i,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 +inf.

- FLOOR res=i desc=f [VH–VL]

  Return Kid 0 rounded towards -inf .

- 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.

- STRCTFLD res=A4,A8 desc=A4,A8 [VH]

  This operator represents a source-level selection operator, except that instead of taking a value (a whole structure) and returning a value (a field in the structure), it takes the address of the structure given by Kid 0 and returns the address of the field reference identified by field_id . (Note that the field_id refers to the *unflattend* structure layout; see below.) Thus, the operator really is semantic sugar for some pointer arithimetic; it performs no loads (ILOADS or LDAs). The operator contains two ty_idx's, one giving the type of the structure (WN_load_addr_ty) and the other the type of the field (WN_ty).

  Originally, even in VH WHIRL, the representation of structure field accesses (i.e., `x.y` in C or `x%y` in Fortran) was acutely inconvenient for source-to-source compiling (e.g., transformations and unparsing). First, field references were usually represented using either the offset or field_id entries of one of the WHIRL load or store operators – but not always. For example, to access a field in an array element, actual OPR_ADD nodes appeared in the AST because OPR_ARRAY has no offset field. Second, even when structure references were represented in WHIRL load operators, they were difficult to understand from a source-level perspective. Consider an access to the first field in a structure. If the front-end did not set the field_id (a la the Fortran front end), then the offset of the first field would be 0 and the only way to detect the presence of the field-selection operator was by comparing the type of the loaded object and the type of base pointer. Consider a second example where the source code references a field in a nested structure, that is, a structure within a structure. Because the typical semantics of a WHIRL load assumed *flattened* structures, the unparsers had to do a lot of work to reconstruct the multiple field-selection operators that actually appeared at the source-code level.

  We designed this operator so that writing a pass to lower it to the original implicit plus flattened representation would be very eady.

- 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 bitfield 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 ALLOCAs: 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.

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

Table 1.3: Examples to show relationship between MOD and REM

- 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 $(imodj)$ is defined as the value of the expression $(i - k * j)$ 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. $-(-i \bmod -j)$ yields the same value as $(i \bmod j)$. 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, $(i \bmod j) = (i\%j) + j$.

- 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.

- 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 evaluates 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 evaluates 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 HWHIRL 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 Kid 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 Kid 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 ofthe 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 * Kid 2) + Kid 0.

- MSUB res=f [VL]

  Return (Kid 1 * Kid 2) - Kid 0.

- NMADD res=f [VL]

  Return - ((Kid 1 * Kid 2) + Kid 0).

- NMSUB res=f [VL]

  Return - ((Kid 1 * Kid 2) - Kid 0).

1.15.4 N-ary Operations

- ARRAY res=A4,A8 [VH–H]

  This operator uses array addressing rules (row-major, zero-based) 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 m1..mn, and if we name the values of the index expressions x1..xn (i.e. xi = the value of Kid i+n), and if element_size is s, then for contiguous arrays, the resultant address is:

$$kid\ 0 + s \sum_{i=1}^{n} \left(x_i \prod_{j=i+1}^{n} m_j \right)$$

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

$$kid\ 0 + (-s) \sum_{i=1}^{n} x_i m_j$$

  In contiguous arrays, for dimensions $d = 2 \ldots n$, $0 \leq x_d < m_d$; 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 BLOCKs 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

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

```
DO_LOOP
  <index var>
INIT
  <initialization statement>
COMP
  <comparison for end condition>
INCR
  <increment statement>
BODY
  BLOCK
    ...
  END_BLOCK


IF
  <condition>
THEN
  BLOCK
    ...
  END_BLOCK
ELSE
  BLOCK
    ...
  END\_BLOCK
END_IF
```

```
FUNC_ENTRY
  IDNAME
  IDNAME
BODY
  BLOCK
    ...
  END_BLOCK


DO_WHILE
  <index var>
BODY
  BLOCK
    ...
  END_BLOCK

WHILE_DO
  <index var>
BODY
  BLOCK
    ...
  END_BLOCK
```

Figure 1.5: ASCII Formats for Structured Control Flow Statements

Chapter 2

Whirl Symbol Table

2.1 Introduction and Overview

This document describes the symbol table portion of the WHIRL file produced and used by the Open64 compiler. A separate document describes the WHIRL intermediate program representation. Section 2.22 contains some helpful programming notes.

The WHIRL symbol table is made up of a series of tables. They are designed for compilation, optimization and storage efficiency. The way the tables are organized closely corresponds to the compiler's view of the symbol table. The model also enhances locality in references to the tables. The WHIRL symbol table is divided into the global part and the local part. The local part is organized by program units (PUs). Figure 2.1 gives a pictorial overview of the WHIRL symbol table as produced by the front-ends. There are different kinds of tables. The tables that can appear in both the global and local part of the symbol table are:

1. ST_TAB – This is the fundamental building block of the symbol table. In general, any symbol with a name occupies an entry in this table. Any constant value that reside in memory (floating point and string constants) also occupies an entry in this table.

2. INITO_TAB – Each entry specifies the initial value(s) of an initialized data object. It in turn refers to one or more entries in the INITV_TAB for initial values of each individual component of the data object.

3. ST_ATTR_TAB – Each entry associates some miscellaneous attributes with an entry in the ST_TAB.

The tables that can only appear in the global part of the symbol table are:

1. PU_TAB – Each entry represents a procedure that appears in the source file as either function prototype or definition.

2. TY_TAB – Each entry represents a distinct type in the program. It in turn refers to the FLD_TAB, TYLIST_TAB, ARB_TAB, or PU_TAB to specify the full structure of each type.

3. FLD_TAB – Each entry specifies a field in a struct type.

4. TYLIST_TAB – Each entry specifies a parameter type in a function prototype declaration.

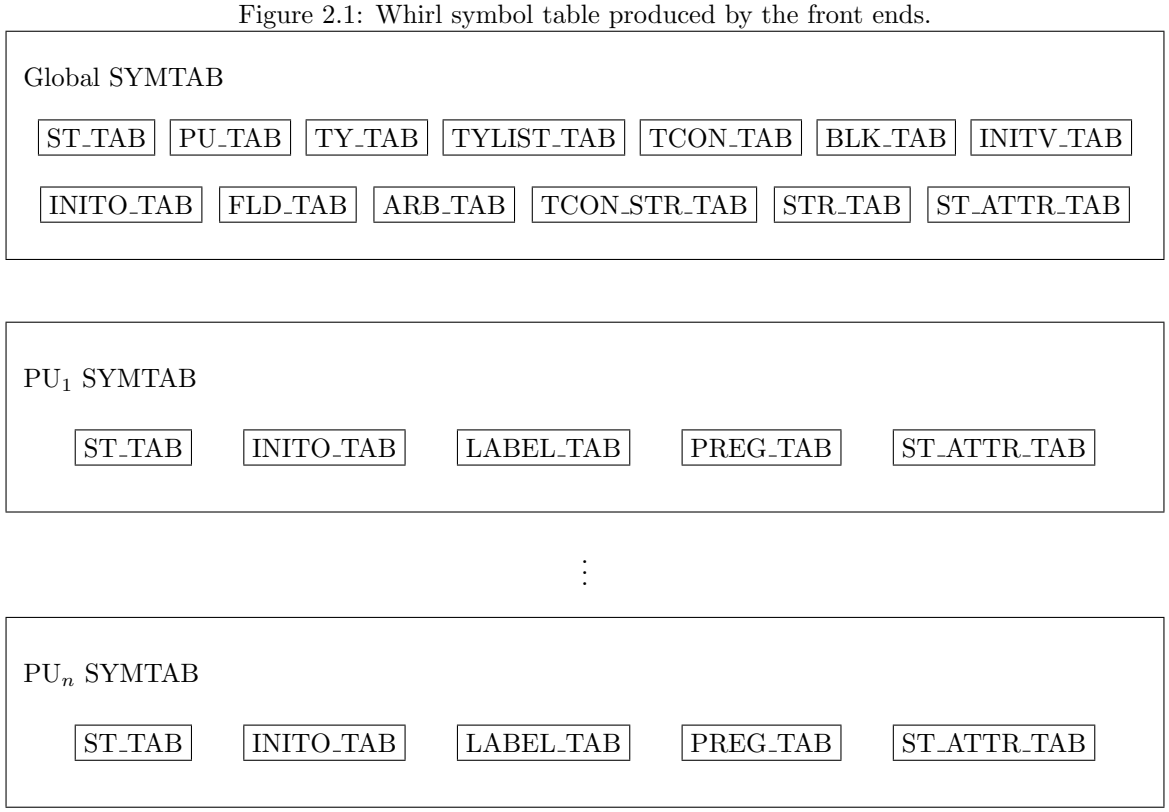
5. ARB_TAB – Each entry gives information about a dimension of an array type.

6. TCON_TAB – The values of any non-integer constants are stored here. For string constants, it in turn refers to the TCON_STR_TAB.

7. BLK_TAB – Each entry specifies layout information of a block of data.

8. INITV_TAB – Each entry describes the initial value of a scalar component of an initialized data object.

Figure 2.1: Whirl symbol table produced by the front ends.



9. STR_TAB – All strings are stored here. They include names of variable, types, labels, etc.

10. TCON_STR_TAB – All string literals defined in the user program are stored in this table.

The tables that can only appear in the local part of the symbol table are:

1. LABEL_TAB –Information associated with each WHIRL label used in the PU is stored here.

2. PREG_TAB –Information associated with each pseudo-register used in the PU is stored here.

Apart from the above tables, each compiler component is free to allocate additional tables for its own internal use in storing extra information. The additional tables are to have the same number of entries and be referred to by the same type of index as one of the above tables. As a general rule, the first entry of each table has index 1; index 0 is reserved to stand for uninitialized index value. The design also assumes that any table will never grow to more than 16 million entries, so that only 24 bits are needed to contain a table index. An exception is STR_TAB, in which the index is really a byte offset.

The tables listed so far mainly serves the purpose of communicating information gathered by the front-ends to the back-end phases during compilation. The back-end optimization phases may create more information, and the new information can reside in additional tables created for the purpose of passing information to the other back-end components. These tables will be prefixed by the name of the component that creates the information in the table, e.g. IPA_ST_TAB, WOPT_ST_TAB, etc. In particular, BE_ST_TAB (Section 2.20.1) serves to communicate information among the back-end components, including IPA.

The remaining sections of this chapter describe the symbol table structures in more details and the interfaces to them.

2.2 SCOPE

Depending on the context, a different set of symbol tables might become visible. For example, in a nested procedure, three ST_TABs are visible –its own local ST_TAB, the parent PU's ST_TAB, and the global ST_TAB. Associated with each PU, a SCOPE array is defined for specifying the list of visible tables. The index to this array is the lexical scope. Index 0 is reserved. Index 1 refers to the global symbol tables, and index 2 refers to the local symbol tables. A nested procedure will have an index starting at 3, depending on the level of nesting. The type of the SCOPE array index is SYMTAB_IDX, which is an unsigned 8-bit integer.

Strictly speaking, SCOPE arrays are not part of the symbol table, and they are never written out to a WHIRL file. Tables that can only appear in the global part of the symbol table are always visible. So they are not explicitly described by the SCOPE array. Each element of a SCOPE array has the following structure, size 24 bytes:

Table 2.1: Layout of a SCOPE Array Element.

Offset	Field	Type	Description	Field size
byte 0	pool	MEM_POOL *	pointer to the memory pool for local tables	1 word
byte 4	st	ST *	pointer to the ST for this PU	1 word
byte 8	st_tab	ST_TAB *	pointer to the table of ST entries	1 word
byte 12	label_tab	LABEL_TAB *	pointer to the table of labels	1 word
byte 16	preg_tab	PREG_TAB *	pointer to the table of pseudo registers	1 word
byte 20	inito_tab	INITO_TAB *	pointer to the table of INITO entries.	1 word
byte 24	st_attr_tab	ST_ATTR_TAB *	pointer to the table of ST_ATTR entries	1 word

For the global scope (i.e., index 1of the SCOPE array), the fields pool, st, label_tab, and preg_tab are not used, and contain the NULL pointer.

2.3 ST_TAB

Each entry of this table is an ST. A symbol in the program is uniquely identified by a value of type ST_IDX.

2.4 ST_IDX

ST_IDX is of size 32 bits, and is composed of two parts:

Table 2.2: Layout of ST_IDX

Field	Description	Field position and size
level	lexical level	least significant 8 bits
index	index to ST_TAB	most significant 24 bits

The low order 8 bits are used to index into the SCOPE array in order to get to the ST_TAB.

2.4.1 ST Entry

The ST entry has the following structure, size 32 bytes:

name_idx/tcon : If sym_class is CLASS_CONST, the tcon field holds the index to the TCON_TAB. For all other sym_class values, the name_idx field holds the index to the STR_TAB. If the export class is EXPORT_LOCAL or EXPORT_LOCAL_INTERNAL, the name is optional. And when there is no name, name_idx should be zero.

flags/flags_ext : Miscellaneous attributes, See Section 2.3.5.

Table 2.3: Layout of ST

Offset	Field	Description	Field size
byte 0	name_idx	STR_IDX to the name string	1 word
byte 0	tcon	TCON_IDX of the constant value	1 word
byte 4	flags	misc. attributes of this entry	1 word
byte 8	flags_ext	more flags for future extension	1 byte
byte 9	sym_class	class of symbol	1 byte
byte 10	storage_class	storage class of symbol	1 byte
byte 11	export	export class of the symbol	1 byte
byte 12	type	TY_IDX of the high-level type	1 word
byte 12	pu	PU_IDX if program unit	1 word
byte 12	blk	BLK_IDX if CLASS_BLOCK	1 word
byte 16	offset	offset from base	2 words
byte 24	base_idx	ST_IDX of the base of the allocated block	1 word
byte 28	st_idx	ST_IDX for this entry	1 word

sym_class : The class of symbol, see Table 2.4.

storage_class : The storage class of symbol, see Table 2.5.

export : The export class of symbol, see Section 2.3.4.

type/pu/blk : If sym_class is CLASS_FUNC, then the pu field holds the index to the PU_TAB. If sym_class is CLASS_BLOCK, this field holds the BLK_IDX. If sym_class is CLASS_NAME, this field must be zero. Forall other valid sym_class values, the type field holds the TY_IDX that describes the type of this symbol.

One exception is a CLASS_NAME symbol that has the ST_ASM_FUNCTION_ST bit set, in which case the pufield holds the index to the PU_TAB.

offset : The byte offset from base_idx. If base_idx is equal to st_idx, then offset must be zero.

base_idx : This is the ST_IDX for the ST that describes the base address (i.e., this symbol is an alias of the specified symbol). If it is equal to it's own st_idx, then the address of this symbol is independently assigned. If ST_IS_WEAK_ALIAS is set, base_idx is overloaded to specify the corresponding strong definition (see Table 2.9 and Section 2.3.7). If ST_IS_SPLIT_COMMON is set, base_idx is overloaded to be the full common definition. It is illegal to set both ST_IS_WEAK_ALIAS and ST_IS_SPLIT_COMMON.

The following rules apply when setting the base address of a symbol. If a symbol A is based on symbol B (i.e. base_idx of A is equal to st_idx of B), then:

1. storage_class of A must be the same as storage_class of B, except when the sym_class of B is CLASS_BLOCK and storage_class of B is SCLASS_UNKNOWN.
2. if sym_class of A is CLASS_BLOCK, sym_class of B must be CLASS_BLOCK.
3. offset of A plus the size of A must not be larger than the size of B.

st_idx : ST_IDX of this symbol. This is used mainly for fast conversion from a pointer to a given ST to the corresponding ST_IDX.

2.4.2 Symbol Class and Storage Class

There is a symbol class and a storage class associated with each ST entry, both of which are enumeration types:

Table 2.4: Symbol Class

Name	Value	Description
CLASS_UNK	0	uninitialized
CLASS_VAR	1	data variable
CLASS_FUNC	2	function
CLASS_CONST	3	constant, a TCON holds the real value
CLASS_PREG	4	pseudo register
CLASS_BLOCK	5	base address for a block of data
CLASS_NAME	6	placeholder for a named ST entry
CLASS_MODULE	7	reserved for module variables but not currently in use
CLASS_TYPE	8	a derived type name
CLASS_PARAMETER	9	a Fortran parameter

Table 2.5: Storage Class

Name	Value	Description
SCLASS_UNKNOWN	0	no specific storage class (e.g., a block of data of mixed storage classes)
SCLASS_AUTO	1	local stack variable
SCLASS_FORMAL	2	formal parameter
SCLASS_FORMAL_REF	3	reference parameter
SCLASS_PSTATIC	4	PU scope static data
SCLASS_FSTATIC	5	file scope static data
SCLASS_COMMON	6	common block (linker allocated)
SCLASS_EXTERN	7	unallocated external data or text
SCLASS_UGLOBAL	8	uninitialized global data
SCLASS_DGLOBAL	9	initialized global data
SCLASS_TEXT	10	executable code
SCLASS_REG	11	register variable
SCLASS_CPLINIT	12	special data object describing initialization of static/global C++ classes.
SCLASS_EH_REGION	13	special table describing C++ exception handling (See Section 2.3.6)
SCLASS_EH_REGION_SUPP	14	supplemental data structure for C++ exception handling (See Section 2.3.6)
SCLASS_DISTR_ARRAY	15	data object that is placed in the special Elf section _MIPS_distr_array
SCLASS_COMMENT	16	names of such symbols are to be placed in the special Elf section comment.
SCLASS_THREAD_PRIVATE_FUNCS	17	data object that is placed in the special Elf section _MIPS_thread_private_funcs
SCLASS_MODULE	18	module symbol (not a common block)

Not all combinations of symbol class and storage class are valid. Only those listed in Table 2.6 are allowed:

Table 2.6: Valid Symbol Class and Storage Class Combinations

Symbol class	Storage class	Description
CLASS_UNK	SCLASS_UNKNOWN	uninitialized
CLASS_VAR	SCLASS_AUTO	stack variable
CLASS_VAR	SCLASS_FORMAL	formal parameter
CLASS_VAR	SCLASS_FORMAL_REF	reference parameter
CLASS_VAR	SCLASS_PSTATIC PU	scope static variable
CLASS_VAR	SCLASS_FSTATIC	file scope variable
CLASS_VAR	SCLASS_COMMON	common block
CLASS_VAR	SCLASS_EXTERN	unallocated external variable
CLASS_VAR	SCLASS_UGLOBAL	uninitialized global variable
CLASS_VAR	SCLASS_DGLOBAL	initialized global variable
CLASS_VAR	SCLASS_CPLINIT	special data object describing initialization of static/global C++ classes.
CLASS_VAR	SCLASS_EH_REGION	special table describing C++exception handling
CLASS_VAR	SCLASS_EH_REGION_SUPP	supplemental data structure for C++ exception handling
CLASS_VAR	SCLASS_DISTR_ARRAY	data object that is placed in the special Elf section _MIPS_distr_array
CLASS_VAR	SCLASS_THREAD_PRIVATE_FUNCS	data object that is placed in the special Elf section _MIPS_thread_private_funcs
CLASS_FUNC	SCLASS_EXTERN	undefined function
CLASS_FUNC	SCLASS_TEXT	defined function
CLASS_CONST	SCLASS_FSTATIC	constant
CLASS_CONST	SCLASS_EXTERN	constant symbol defined in another file (e.g. in IPA-generated symbol table)
CLASS_PREG	SCLASS_REG	pseudo register CLASS_BLOCK all storage classes except SCLASS_UNKNOWN and SCLASS_REG
CLASS_BLOCK	SCLASS_UNKNOWN	a block of data or text of unspecified storage class (e.g., a block of mixed storage classes)
CLASS_NAME	SCLASS_UNKNOWN	an ST entry that only has a name and nothing else, usually used as a placeholder for special symbols that are passed to the linker
CLASS_NAME	SCLASS_COMMENT	an ST entry whose name is to be placed in the Elf section .comment

2.4.3 Export Scopes

This enumeration describes the possible scopes that symbols exported from a file may map into, i.e., linker globals for DSO (dynamically shared object)-related components.

Only an EXPORT_LOCAL or EXPORT_LOCAL_INTERNAL symbol must be defined in the file being compiled. All others can be either defined or undefined. All symbols except EXPORT_PREEMTIBLE must be defined in the current DSO or executable.

Only EXPORT_LOCAL and EXPORT_LOCAL_INTERNAL symbols are allowed in a local ST_TAB. Symbols with all other export scopes must be placed in the global ST_TAB. Furthermore, the ST entries of all functions, regardless of export scope, must be placed in the global ST_TAB.

Table 2.7: Export Scopes

Export Scope	Value	Description

Table 2.7: Export Scopes

Export Scope	Value	Description
EXPORT_LOCAL	0	not exported, must be defined in current file (e.g. C static data), address can be exported from DSO using a pointer
EXPORT_LOCAL_INTERNAL	1	not exported, must be defined in current file, only visible within current file, only used within the DSO or executable
EXPORT_INTERNAL	2	exported, only visible and used within the DSO or executable, must be defined in current DSO or executable
EXPORT_HIDDEN	3	exported, name is hidden within DSO or executable, address can be exported from DSO using a pointer, must be defined in current DSO or executable
EXPORT_PROTECTED	4	exported, non-preemptible, must be defined in current DSO or executable
EXPORT_PREEMPTIBLE	5	exported, preemptible
EXPORT_OPTIONAL	6	correspond to STO_OPTIONAL in Elfsymbol table (see `<sys/elf.h>`)

Valid combinations of export scopes and storage classes are listed in the following table:.

Table 2.8: Valid Combinations of Storage Class and Export Scopes

Storage class	Export scopes	Description
SCLASS_UNKNOWN SCLASS_AUTO SCLASS_FORMAL SCLASS_FORMAL_REF SCLASS_PSTATIC SCLASS_FSTATIC SCLASS_CPLINIT SCLASS_EH_REGION SCLASS_EH_REGION_SUPP SCLASS_DISTR_ARRAY SCLASS_THREAD_PRIVATE_FUNCS SCLASS_COMMENT	EXPORT_LOCAL EXPORT_LOCAL_INTERNAL EXPORT_INTERNAL EXPORT_HIDDEN EXPORT_PROTECTED EXPORT_PREEMPTIBLE	DSO scope data or text symbols
SCLASS_COMMON SCLASS_DGLOBAL	EXPORT_LOCAL EXPORT_LOCAL_INTERNAL	member of a common or data block; these symbols must have base_idx pointing to an ST entry with the same storage class
SCLASS_EXTERN	EXPORT_LOCAL EXPORT_LOCAL_INTERNAL	local symbols that are not defined in the current file; use in IPA-generated file where a CLASS_CONST symbol is defined in a separate file.
SCLASS_TEXT	EXPORT_LOCAL EXPORT_LOCAL_INTERNAL	static functions
SCLASS_TEXT	EXPORT_INTERNAL EXPORT_HIDDEN EXPORT_PROTECTED EXPORT_PREEMPTIBLE	global functions
SCLASS_REG	EXPORT_LOCAL EXPORT_LOCAL_INTERNAL	registers

2.4.4 ST Flags

Associated with each ST entry are one or more attributes that describe specific property of it. Some of them are mutually exclusive and some of them are related. They are described in the following table:

Table 2.9: Miscellaneous Attributes of an ST Entry Flag/Value Description

Symbol	Description
ST_IS_WEAK_SYMBOL 0x00000001	weak name • not valid for EXPORT_LOCAL or EXPORT_LOCAL_INTERNAL • see Section 2.3.7 for semantics of weak symbols
ST_IS_SPLIT_COMMON 0x00000002	part of a split common • base_idx gives the ST_IDX of the corresponding complete common definition • ST_IS_WEAK_SYMBOL must not be set ST_IS_NOT_USED0x00000004 symbol is not referenced
ST_IS_INITIALIZED 0x00000008	initialized static or global variable • only valid for CLASS_VAR, CLASS_CONST, and CLASS_BLOCK • only valid for SCLASS_PSTATIC, SCLASS_FSTATIC, SCLASS_EXTERN, SCLASS_DGLOBAL, SCLASS_UGLOBAL, SCLASS_CPLINIT, SCLASS_EH_REGION, SCLASS_EH_RGION_SUPP, SCLASS_DIST_ARRAY, and SCLASS_THREAD_PRIVATE_FUNCS. • also valid for SCLASS_UNKNOWN if symbol class is CLASS_BLOCK • for SCLASS_UGLOBAL, ST_INIT_VALUE_ZERO must be set(uninitialized globals and globals explicitly initialized to zero are equivalent) • must be set for SCLASS_DGLOBAL • for CLASS_VAR, if ST_INIT_VALUE_ZERO is not set, there must be a corresponding INITO entry
ST_IS_RETURN_VAR 0x00000010	return value for Fortran function • only valid for SCLASS_AUTO
ST_IS_VALUE_PARM 0x00000020	parameter is passed by value • only valid for SCLASS_FORMAL

Table 2.9: Miscellaneous Attributes of an ST Entry Flag/Value Description

Symbol	Description
ST_PROMOTE_PARM 0x00000040	parameter has been promoted from chars/short to int or from float to double • only valid for C/C++
ST_KEEP_NAME_W2F 0x00000080	whirl2f should neither declare nor rename this symbol • only valid for CLASS_VAR
ST_IS_DATAPOOL 0x00000100	Fortran data pools
ST_IS_RESHAPED 0x00000200	symbol has a distribute_reshape pragma supplied for it; only valid for CLASS_VAR
ST_EMIT_SYMBOL 0x00000400	must appear in the symbol table of the Elf object file • only valid for CLASS_VAR, CLASS_NAME, and CLASS_FUNC, • used by C++ to force certain local symbols to be written out to the Elf object file
ST_HAS_NESTED_REF 0x00000800	symbol is referenced by a PU nested in the current PU • only valid for SCLASS_AUTO, SCLASS_PSTATIC, SCLASS_FORMAL, and SCLASS_FORMAL_REF.
ST_INIT_VALUE_ZERO 0x00001000	uninitialized global or static symbol • only valid for CLASS_VAR • only valid for SCLASS_EXTERN, SCLASS_UGLOBAL, SCLASS_FSTATIC, and SCLASS_PSTATIC • ST_IS_INITIALIZED must be set • also valid for symbol explicitly initialized to zero
ST_GPREL 0x00002000	can be accessed via an offset from the global pointer • only valid for CLASS_VAR and CLASS_CONST • not valid for SCLASS_AUTO, SCLASS_FORMAL, and SCLASS_FORMAL_REF
ST_NOT_GPREL 0x00004000	can not be accessed via an offset from the global pointer • only valid for CLASS_VAR and CLASS_CONST • not valid for SCLASS_AUTO, SCLASS_FORMAL, and SCLASS_FORMAL_REF

Table 2.9: Miscellaneous Attributes of an ST Entry Flag/Value Description

Symbol	Description
ST_IS_NAMELIST 0x00008000	special symbol for namelists • only valid for CLASS_VAR • used by whirl2f to identify namelist symbols
ST_IS_F90_TARGET 0x00010000	symbol may be accessed by dereferencing an F90 pointer • only valid for CLASS_VAR • if not set, no direct load or store to this symbol can alias with any load or store through an F90 pointer • if not set, no indirect load or store through an F90 pointer can access this item
ST_DECLARED_STATIC 0x00020000	VMS formals declared static • only valid for CLASS_VAR
ST_IS_EQUIVALENCED 0x00040000	part of an Fortran equivalence • only valid for CLASS_VAR
ST_IS_FILL_ALIGN 0x00080000	symbol has a fill_symbol or align_symbol pragma supplied • only valid for CLASS_VAR
ST_IS_OPTIONAL_ARGUMENT 0x00100000	formal parameter is optional • only valid for SCLASS_FORMAL and SCLASS_FORMAL_REF • it is illegal to speculate loads/stores of this symbol

Table 2.9: Miscellaneous Attributes of an ST Entry Flag/Value Description

Symbol	Description
ST_PT_TO_UNIQUE_MEM 0x00200000	memory location pointed to by this symbol cannot be accessed via any other way • only valid for SCLASS_VAR • only valid for pointer, or non-scalar type that contains pointers • only valid for compiler-generated symbols • for non-scalar type, such as a struct that contains a pointer or an array of pointers, this flag applies to all pointers within the structure • a pointer with this bit set refers to a memory location that is never accessed indirectly via any other pointer or directly via any local or global variable in the entire program • the compiler phase that sets this bit must guarantee that the above property holds even through inlining or other code motion • copying such pointers to another pointers is allowed, as long as • these other pointers are never dereferenced
ST_IS_TEMP_VAR 0x00400000	compiler generated temporary variable or formal parameters • only valid for SCLASS_AUTO, SCLASS_FORMAL, and SCLASS_FORMAL_REF
ST_IS_CONST_VAR 0x00800000	read-only static or global variable • only valid for CLASS_VAR • not valid for SCLASS_AUTO, SCLASS_FORMAL, and SCLASS_FORMAL_REF • compiler can allocate this symbol in read-only data segment
ST_ADDR_SAVED 0x01000000	the address of this symbol is saved to another variable • not valid for SCLASS_REG
ST_ADDR_PASSED 0x02000000	the address of this symbol is passed to another PU as actual parameter • not valid for SCLASS_REG • this flag is now re-computed by the compiler backend and is not set by the frontend

Table 2.9: Miscellaneous Attributes of an ST Entry Flag/Value Description

Symbol	Description
ST_IS_THREAD_PRIVATE 0x04000000	symbol is a private data object of an MP program • storage of this symbol is not shared by the threads of an MP program
ST_PT_TO_COMPILER_GENERATED_MEM 0x08000000	symbol is a pointer to compiler-allocated memory space • only valid for pointer type • only valid for compiler-generated symbols • pragmas or other data object attributes specified by users do not apply to this memory location because it is not visible to them
ST_IS_SHARED_AUTO 0x10000000	an automatic variable that is accessed within a parallel region and has shared scope • only valid for SCLASS_AUTO
ST_ASSIGNED_TO_DEDICATED_PREG 0x20000000	symbol is associated to a dedicated (hardware) register • compiler should always keep this symbol's value in the specified register • only valid for CLASS_VAR • must be volatile type
ST_ASM_FUNCTION_ST 0x40000000	name of this symbol is an assembly language code corresponding to a program unit • only valid for symbols in the global symbol table • only valid for CLASS_NAME, SCLASS_UNKNOWN • only valid for EXPORT_LOCAL • not valid for nested PU • must have valid PU_IDX, and the corresponding PU entry must have PU_NO_DELETE and PU_NO_INLINE bits set, with a 0 TY_IDX.

2.4.5 Exception Handling Region

Symbols of storage class SCLASS_EH_REGION are allocated by the code generator for the tables that control exception-handling. These tables are allocated in a special section created by the linker; they never correspond directly to program entities. They have no existence before code generation, so they are never referred to in the WHIRL. Symbols of storage class SCLASS_EH_REGION_SUPP represent initialized variables created by the frontend to provide supplementary information about exception-handling actions

to be taken by the exception-handling runtimes when an exception is thrown. They are allocated in a second special section created by the linker. They appear in the ereg_suppfield of the WHIRL, but only the exception-handling part of the code generator should ever look at them.

The data in the sections corresponding to the storage class SCLASS_EH_REGION and the storage class SCLASS_EH_REGION_SUPP should be readonly by the exception-handling runtimes and should never be modified once it is generated. Symbols of storage class SCLASS_EH_REGION or SCLASS_EH_REGION_SUPP have a very unique semantic with respect to storage and scope. They are local to the PU in terms of scope, meaning that they can only be referenced from within the defining PU. Their storage is not allocated form the stack, but from the global storage area.

Hence, multiple instances of the same PU (e.g., recursive calls) share the same memory locations and values of these symbols. However, they differ from SCLASS_PSTATIC symbols in that when the defining PU is cloned or inlined, new copies of these symbols need to be created.

2.4.6 Semantics of Weak Symbols

The semantics of a weak symbol depends on its storage_class and base_idx, which is summarized in the following table:

Table 2.10: Semantics of Weak Symbols

storage_class	base_idx != st_idx	base_idx == st_idx
SCLASS_TEXT SCLASS_UGLOBAL SCLASS_DGLOBAL	weak symbol that has storage allocated (See note 1)	weak definition before data layout (See note 2)
SCLASS_EXTERN	weak symbol with an alias to a strong definition (See note 3)	undefined weak symbol (See note 4)

1. This refers to defined variables or functions that are marked weak. After layout, they can be based on other symbols. The weak flag means that they can be preempted by a strong definition. When they are preempted, their associated storage is either wasted or can be deleted.

2. Similar to (1), with the exception that storage of this symbol has not been laid out. Basically, treat (1) and (2) as regular variable or function definitions, with the exception that they might be preempted by a strong definition. Once preempted, they corresponding storage cannot be referenced via this symbol name.

3. This is a weak alias to a strong definition. The name of this symbol is bound with the storage owned by the corresponding strong definition (specified by base_idx). The weak attribute makes this binding preemptible.

4. This symbol has no storage of its own and is not associated with any other symbol. The linker should not complain when no definition can be found, and should assign 0 as its address.

2.5 PU_TAB

Each entry of this table gives information about each PU that appears in the source file either as procedure declaration or function prototype. The index to this table, PU_IDX, can be used as a PU identifier. The PU entry has the following structure, size 24 bytes:

Table 2.11: Layout of PU

Offset	Field	Description	Field size
byte 0	target_idx	TARGET_INFO_IDX to the target-specific info.	1 word
byte 4	prototype	TY_IDX to give the prototype type information	1 word
byte 8	lexical_level	lexical level (scope) of symbols in this PU	1 byte
byte 9	gp_group	gp-group number of this PU	1 byte
byte 10	src_lang	source language of this PU	1 byte
byte 11	unused	unused, must be filled with zeros.	5 bytes
byte 16	flags	flags associated with this function prototype	2 words

target_idx : Index to TARGET_INFO_TAB, which contains the target-specific information about this PU such as register usage information, etc. The TARGET_INFO_TAB is current undefined and is reserved for future expansion. In the current release, target_idx must be zero.

prototype: The TY_IDX for the type of the function.

lexical_level: Lexical level of symbols defined in this PU (i.e. index to the SCOPE array, see Section 2.2). It is always greater than 1.

gp_group : Gp-group id for this PU; used in multi-got program. Single GOT programs have gp_group zero.

src_lang : Source language of this PU, see Table 2.13.

unused: For alignment of flags, must be filled with zeros.

flags: Miscellaneous attributes, see Table 2.12.

Table 2.12: Miscellaneous Attributes of a PU

Entry	Flag/Value	Description
PU_IS_PURE	0x00000001	pure function • does not modify the global state • does not make reference to the global state
PU_NO_SIDE_EFFECTS	0x00000002	does not modify the global state
PU_IS_INLINE_FUNCTION	0x00000004	inline keyword specified • function may be inlined
PU_NO_INLINE	0x00000008	function must not be inlined • mutually exclusive with PU_MUST_INLINE
PU_MUST_INLINE	0x00000010	function must be inlined • mutually exclusive with PU_NO_INLINE
PU_NO_DELETE	0x00000020	function must never be deleted
PU_HAS_EXC_SCOPES	0x00000040	has C++ exception handling region, or would have if exceptions were enabled. • PU_CXX_LANG must be set

Table 2.12: Miscellaneous Attributes of a PU

Entry	Flag/Value	Description
PU_IS_NESTED_FUNC	0x00000080	a nested function • lexical_level must be larger than 2
PU_HAS_NON_MANGLED_CALL	0x00000100	function is called with non-reshaped array as actual parameter • must keep a copy of the function with non-mangled name
PU_ARGS_ALIASED	0x00000200	parameters might point to same or overlapping memory location • PU_F77_LANG or PU_F90_LANG must be set PU_NEEDS_FILL_ALIGN_LOWERING 0x00000400 contains symbols that have the fill_symbol or align_symbol pragma specified
PU_NEEDS_T9	0x00000800	register $t9 must contain the lowest address of the PU
PU_HAS_VERY_HIGH_WHIRL	0x00001000	PU has very high WHIRL
PU_HAS_ALTENTRY	0x00002000	PU contains alternate entry points • PU_F77_LANG or PU_F90_LANG must be set
PU_RECURSIVE	0x00004000	PU is self-recursive, or is part of a multi-function recursion
PU_IS_MAINPU	0x00008000	main entry point of a program
PU_UPLEVEL	0x00010000	other PU nested in this one
PU_MP_NEEDS_LNO	0x00020000	must invoke LNO on this PU, regardless of compilation options
PU_HAS_ALLOCA	0x00040000	contains calls to alloca
PU_IN_ELF_SECTION	0x00080000	the code generator must put this PU in its own Elfsection
PU_HAS_MP	0x00100000	contains a MP construct
PU_MP	0x00200000	a PU created by the MP lowerer
PU_HAS_NAMELIST	0x00400000	has namelist declaration • PU_F77_LANG or PU_F90_LANG must be set
PU_HAS_RETURN_ADDRESS	0x00800000	contain references to the special symbol __return_address
PU_HAS_REGION	0x01000000	PU has regions in it
PU_HAS_INLINES	0x02000000	PU has inlined code in it
PU_CALLS_SETJMP	0x04000000	PU contains calls to setjmp.
PU_CALLS_LONGJMP	0x08000000	PU contains calls to longjmp.

Table 2.12: Miscellaneous Attributes of a PU

Entry	Flag/Value	Description
PU_IPA_ADDR_ANALYSIS	0x10000000	the ST_ADDR_SAVED bits for all symbols referenced in this PU are set by IPA's address analysis • the compiler backend should trust the (more accurate) results of IPA and need not recompute the ST_ADDR_SAVED bits for this PU
PU_SMART_ADDR_ANALYSIS	0x20000000	suppress the conservative address-taken validation • do not perform conservative address-taken verification, which might set the ST_ADDR_SAVED bit unnecessarily • set when more accurately address analysis has been performed.
PU_HAS_GLOBAL_PRAGMAS	0x40000000 0x80000000	obsolete a dummy PU that contains global pragmas • a place holder for all global scope pragmas
PU_HAS_USER_ALLOCA	0x100000000	PU contains user-specified call to alloca() • if this pu is inlined, an explicitly deallocation needs to be generated
PU_HAS_UNKNOWN_CONTROL_FLOW	0x200000000	PU has control flow going in or out of the puscope that do not following calling convention • tail-call optimization should be disabled

Each entry of this table is a TY. Any high level type in the program is uniquely identified by a value of type TY_IDX.

Table 2.13: Source Language of a PU

Flag	Value	Description
PU_UNKNOWN_LANG	0x00	Source language unknown
PU_MIXED_LANG	0x01	PU contains code from multiple source languages • resulted from cross-file inlining
PU_C_LANG	0x02	Source language is C
PU_CXX_LANG	0x04	Source language is C++
PU_F77_LANG	0x08	Source language is Fortran 77
PU_F90_LANG	0x10	Source language is Fortran 90
PU_JAVA_LANG	0x20	Source language is Java

2.5.1 TY_IDX

TY_IDX is of size 32 bits, and is composed of two parts. The high order 24 bits is the index to TY_TAB. The low order 8 bits contains information that qualifies the type. Among the low order 8 bits is the alignment information. The actual alignment is given by 2^{align}.

Table 2.14: Layout of TY_IDX.[1]

Offset	Field	Description	Field size
bit 0	align	alignment	5 bits
bit 5	const	const type qualifier	1 bit
bit 6	volatile	volatile type qualifier	1 bit
bit 7	restrict	restrict type qualifier	1 bit
bit 8	index	index to TY_TAB	24 bits

2.6 TY_TAB

TY_IDX appears appear in many different places:

1. in WHIRL nodes that access data objects.
2. in ST entries.
3. in components for type specification: TY, FLD, TYLIST.

Each TY has a natural (and maximum) alignment, which can be determined by analysis of the details of the type. Thus, we omit the natural alignment information from the TY. The alignment of a TY directly affects the alignment in the TY_IDX of an object that encloses or refers to it, unless the object's own alignment is modified by pragmas or type casts. An optimization phase may also improve the alignment of an object by forcing better placement during data layout, in which case it only needs to fix up the alignment of the ST's TY_IDX. Whenever the alignment in the TY_IDX of the WHIRL node and the TY_IDX of the ST being accessed by the WHIRL node do not agree, code generation picks the more efficient(better) alignment of the two. Thus, if a phase worsens the alignment of an object, it has to fix the TY_IDX in all the WHIRL references to it, which is normally impossible.

The above rule dealing with alignment also applies to the other type qualifying bits: whenever a type qualifying bit is different between the TY_IDX of the WHIRL node and the TY_IDX of the ST being accessed by the WHIRL node, code generation picks the more efficient of the two.

2.6.1 TY entry

The TY entry has the following structure, size 24 bytes:

size : The size of the type in bytes. For KIND_FUNCTION and KIND_VOID, the size is zero. For KIND_ARRAY, this is the size of the entire array, except when for variable length arrays, the size is zero.

kind : Field describing if the type is a scalar, structure, etc. See Table 2.16.

mtype : WHIRL data type, see Table 2.17. See Table 2.20 for valid combinations of mtype and kind.

flags : Miscellaneous attributes, see Table 2.18.

[1] Bit offsets assume big Endian bit ordering. For example, the index field is always the most significant 24 bits, regardless of the Endianess of the machine.

Table 2.15: Layout of TY

Offset	Field	Description	Field size
byte 0	size	size of the type in bytes	2 words
byte 8	kind	kind of type	1 byte
byte 9	mtype	corresponding WHIRL data type	1 byte
byte 10	flags	TY flags	2 bytes
byte 12	fld	FLD_IDX for struct/class field information	1 word
byte 12	tylist	TYLIST_IDX for function prototype	1 word
byte 12	arb	ARB_IDX for array bound description	1 word
byte 16	name_idx	STR_IDX to the name string	1 word
byte 20	etype	TY_IDX of array element (array only)	1 word
byte 20	pointed	TY_IDX of the pointed-to type (pointers only)	1 word
byte 20	pu_flags	function-specific attributes	1 word

Table 2.16: Kinds of TY

Name	Value	Description
KIND_INVALID	0	invalid or uninitialized
KIND_SCALAR	1	integer or floating point, no kids
KIND_ARRAY	2	array, arb_idx points to array bound description, etype gives the type of the array element
KIND_STRUCT	3	structure or union, fld_idx points to the field description
KIND_POINTER	4	pointers, pointed gives the type that it points to
KIND_FUNCTION	5	function or procedure, tylist_idx points to the list of TY_IDX for the return type and parameter types.
KIND_VOID	6	C void type, no kids

Table 2.17: WHIRL Basic Data Type

Flag	Value	Description
MTYPE_FQ	15	SGI long double
MTYPE_M	16	memory chunk, for structures
MTYPE_C4	17	32-bit complex
MTYPE_C8	18	64-bit complex
MTYPE_CQ	19	128-bit complex
MTYPE_V	20	void type
MTYPE_BS	21	bits
MTYPE_A4	22	32-bit address
MTYPE_A8	23	64-bit address
MTYPE_C10	24	80-bit IEEE complex
MTYPE_C16	25	128-bit IEEE complex
MTYPE_I16	26	128-bit signed integer
MTYPE_U16	27	128-bit unsigned integer

fld/tylist/arb : Index to one of the tables that provide additional type information, depending on the value of kind (see Tabler 2.16). For KIND_SCALAR, KIND_POINTER and KIND_VOID, this field is zero.

name_idx : The name of the type. For anonymous types, this field should be zero.

etype/pointed/pu_flags : For KIND_ARRAY, etype gives the type of the array element. For

Table 2.18: Miscellaneous Attributes of a TY

Entry	Flag/Value	Description
TY_IS_CHARACTER	0x0001	Fortran character type
TY_IS_LOGICAL	0x0002	Fortran logical type
TY_IS_UNION	0x0004	type is a union; only valid for KIND_STRUCT
TY_IS_PACKED	0x0008	struct or class is packed
TY_PTR_AS_ARRAY	0x0010	treat pointer as array (used by whirl2c/whirl2f)
TY_ANONYMOUS	0x0020	anonymous struct/class/union; only valid for KIND_STRUCT
TY_SPLIT	0x0040	split from a larger common block
TY_IS_F90_POINTER	0x0080	pointer is subject to F90 alias rules
TY_NOT_IN_UNION	0x0100	type cannot be part of a union

KIND_POINTER, pointed gives the type that it points to. For KIND_FUNCTION, pu_flags contains attributes of the function. For all other values of kind, this field is zero.

Types that are structurally identical can share common TY entries in order to minimize the size of TY_TAB.

Table 2.19: Attributes of a Function

Flag	Value	Description
TY_NO_ANSI_ALIAS	0x0200	ANSI alias rules do not apply
TY_IS_NON_POD	0x0400	a C++ non-pod structure • constructor/destructor calls must be generated when creating a temp. variable of this type (usually done by the frontend)
TY_RETURN_TO_PARAM	0x00000001	a function returning a struct that is larger than twice the size of the largest integer type • an additional argument (first) is passed which contains the address where the return value is to be placed
TY_IS_VARARGS	0x00000002	allows variable number of arguments • the last formal parameter is a descriptor of the variable part of the parameter list
TY_HAS_PROTOTYPE	0x00000004	function has ANSI-style prototype defined.

2.7 FLD_TAB

Each entry of this table gives information about a field in a struct or union. The TY of the struct type points to the FLD entry for the first field. The remaining fields follow in consecutive FLD_TAB entries until a flag indicates it is the last field. The FLD entry has the following structure, size 24 bytes:

name_idx : STR_IDX to the name string, 0 if anonymous.

type: The TY_IDX of this field. If ofst is equal to the total sizeof the struct, the size of the type pointed to by type must be zero.

Table 2.20: Valid Combinations of TY Kinds and WHIRL Data Types

Kind	Valid WHIRL data type
KIND_SCALAR	all mtypes except MTYPE_UNKNOWN and MTYPE_V
KIND_ARRAY	MTYPE_UNKNOWN and MTYPE_M
KIND_STRUCT	MTYPE_M
KIND_POINTER	MTYPE_U4 or MTYPE_U8 (for MIPS) MTYPE_A4 or MTYPE_A8 (for Merced)
KIND_FUNCTION	MTYPE_UNKNOWN
KIND_VOID MTYPE_V	

Table 2.21: Layout of FLD

Offset	Field	Description	Field size
byte 0	name_idx	STR_IDX to the name string	1 word
byte 4	type	TY_IDX of field	1 word
byte 8	ofst	offset within struct in bytes	2 words
byte 16	bsize	bit field size in bits	1 byte
byte 17	bofst	bit field offset starting at byte specified by	1 byte
byte 18	flags	FLD flags	2 bytes
byte 20	st	ST_IDX to the ST entry, if any.	4 bytes

Table 2.22: Miscellaneous Attributes of an FLD Entry

Flag	Value	Description
FLD_LAST_FIELD	0x0001	indicate the last field in a struct
FLD_EQUIVALENCE	0x0002	this field belongs to an equivalence of a common block (i.e., overlaps in memory with other common block element(s))
FLD_BEGIN_UNION	0x0004	beginning of a union in a Fortran record
FLD_END_UNION	0x0008	end of a union in a Fortran record
FLD_BEGIN_MAP	0x0010	beginning of a map in a Fortran record
FLD_END_MAP	0x0020	end of a map in a Fortran record
FLD_IS_BIT_FIELD	0x0040	indicate a bit field • bsize and bofst are valid only if this flag is set

ofst : The byte offset of this field within the struct. This must be less than or equal to the total size of the struct.

When the offset is equal to the size of the struct, type must be an TY_IDX of a type with zero size.

bsize : The size of the bit field in number of bits. Valid only if FLD_IS_BIT_FIELD is set; must be zero otherwise.

bofst : The bit field offset starting at the byte specified by ofst. Valid only if FLD_IS_BIT_FIELD is set; must be zero otherwise.

flags : Miscellaneous attributes, see Table 2.22.

st : ST_IDX to the (optional) ST entry corresponding to this field.

- typically used for common block elements where each element has a separate ST entry.
- the ST entry must be one in the global symbol table.
- when not set, must be zero.

2.8 TYLIST_TAB

Each entry of this table gives the type of each parameter in a function prototype. The TY of the function prototype points to the TYLIST entry that gives the return type. The ensuing entries give the types of the parameters. A TY_IDX value of 0 specifies the end of the parameter list. The TYLIST entry has the following structure:

Table 2.23: Layout of TYLIST

Offset	Field	Description	Field size
byte 0	type	TY_IDX to the type	1 word

2.9 ARB_TAB

Each entry of this table gives information about a dimension of an array. The TY of the array type points to the ARB entry for the first dimension, indicated by ARB_FIRST_DIMEN. For C/C++ arrays, this corresponds to the leftmost dimension. For Fortran arrays, this corresponds to the right-most dimension. The remaining dimensions follow in consecutive ARB_TAB entries until an entry with ARB_LAST_DIMEN set. The dimension of the array must be specified in dimension of every entry.

The ARB entry has the following structure, size 32 bytes:

Table 2.24: Layout of ARB

Offset	Field	Description	Field size
byte 0	flags	misc. attributes	2 bytes
byte 2	dimension	dimension of the array	2 bytes
byte 4	unused	unused, must be filled with zeros	1 word
byte 8	lbnd_val	constant lower bound value	2 words
byte 8	lbnd_var	ST_IDX of variable that stores the non-constant lower bound	1 word
byte 12	lbnd_unused	filler for lbnd_var, must be zero	1 word
byte 16	ubnd_val	constant upper bound value	2 words
byte 16	ubnd_var	ST_IDX of variable that stores the non-constant upper bound	1 word
byte 20	ubnd_unused	filler for ubnd_var, must be zero	1 word
byte 24	stride_val	constant stride	2 words
byte 24	stride_var	ST_IDX of variable that stores the non-constant stride	1 word
byte 28	stride_unused	filler for stride_var, must be zero	1 word

Table 2.25: Miscellaneous Attributes of an ARB Entry

Flag	Value	Description
ARB_CONST_LBND	0x0001	lower bound is constant
ARB_CONST_UBND	0x0002	upper bound is constant
ARB_CONST_STRIDE	0x0004	stride is constant
ARB_FIRST_DIMEN	0x0008	current dimension is first
ARB_LAST_DIMEN	0x0010	current dimension is last

2.10 TCON_TAB

Each entry of this table is the TCON for storing integer, floating point or string constant values. The first three entries of this table are reserved. The first entry (index 0) is reserved for uninitialized index value. The second entry (index 1) always contains 4-byte floating point value 0.0. the third entry (index 2) always contains 8-byte floating point value 0.0. These entries are shared. All other values are entered independently without checking for duplicates. The TCON entry has the following structure, size 40 bytes:

Table 2.26: Layout of TCON

Offset	Field	Description	Field size
byte 0	ty	WHIRL data type, see Table 2.17	1 word
byte 4	flags	misc. attributes	1 word
byte 8	ival	signed integer (MTYPE_I1, MTYPE_I2, and MTYPE_I4)	1 word
byte 8	uval	unsigned integer (MTYPE_U1, MTYPE_U2, and MTYPE_U4)	1 word
byte 8	i0	64-bit signed integer (MTYPE_I8)	2 words
byte 8	k0	64-bit unsigned integer (MTYPE_U8)	2 words
byte 8	fval	32-bit floating point (MTYPE_F4)real part for 32-bit complex (MTYPE_C4)	1 word
byte 8	dval	64-bit floating point (MTYPE_F8)real part for 64-bit complex (MTYPE_C8)	2 words
byte 8	qval	128-bit floating point (MTYPE_FQ)real part for 128-bit complex (MTYPE_CQ)	4 words
byte 8	sval	string literal (MTYPE_STR/MTYPE_STRING) • byte 8 holds a character pointer (1 word) • byte 12 holds the number of bytes of the string (1 word)	3 words
byte 24	fival	imaginary part for 32-bit complex(MTYPE_C4)	1 word
byte 24	dival	imaginary part for 64-bit complex(MTYPE_C8)	2 words
byte 24	qival	imaginary part for 128-bit complex(MTYPE_CQ)	4 words

2.11 INITO_TAB

Each entry of this table connects an initialized global or static data object with an INITV entry (see Section 2.11), which describes the initial values. Each entry of this table is an INITO, which is identified by a value of type INITO_IDX.

2.11.1 INITO_IDX

INITO_IDX has an identical structure as a ST_IDX. It is of size 32 bits, and is composed of two parts:

Table 2.27: Layout of INITO_IDX

Field	Description	Field position and size
level	lexical level	least significant 8 bits
index	index to INITO_TAB	most significant 24 bits

The low order 8 bits are used to index into the SCOPE array in order to get to the INITO_TAB.

2.11.2 INITO Entry

The INITO entry has the following structure, size 8 bytes:

Table 2.28: Layout of INITO

Offset	Field	Description	Field size
byte 0	st_idx	ST_IDX of the variable to be initialized	1 word
byte 4	val	INITV_IDX of the initial values description	1 word

2.12 INITV_TAB

Each entry of this table specifies the initial value of a scalar component of a data object. Initial values of complex data objects are described by a tree of INITV entries, the root of which specified by the INITV_IDX of an INITO.

The INITV entry has the following structure, size 16 bytes:

Table 2.29: Layout of INITV

Offset	Field	Description	Field size
byte 0	next	INITV_IDX for the value of the next array element or the field in a struct	1 word
byte 4	kind	kind of the INITV, see Table 2.30.	2 bytes
byte 6	repeat1	repeat factor except for INITVKIND_VAL	2 bytes
byte 8	st	ST_IDX of symbol for INITVKIND_SYMOFF	1 word
byte 8	lab	LABEL_IDX of symbol for INITVKIND_LABEL	1 word
byte 8	lab1	LABEL_IDX of label for INITVKIND_SYMDIFF(16)	1 word
byte 8	mtype	WHIRL data type for INITVKIND_ZERO and INITVKIND_ONE	1 word
byte 8	tc	TCON_IDX for INITVKIND_VAL	1 word
byte 8	blk	INITV_IDX for INITVKIND_BLOCK	1 word
byte 8	pad	padding in bytes	1 word
byte 12	ofst	byte offset from st for INITVKIND_SYMOFF	1 word
byte 12	st2	ST_IDX of symbol for INITVKIND_SYMDIFF(16)	1 word
byte 12	repeat2	repeat factor for INITVKIND_ZERO, INITVKIND_ONE, and INITVKIND_VAL	1 word
byte 12	unused	filler for INITVKIND_BLOCK, INITVKIND_PAD, and INITVKIND_LABEL, must be zero	1 word

next/blk : The values of a data object are specified by a tree of INITVs, with the root of the tree pointed to by the INITO. INITVs specifying scalars are linked up by the next field, each of which contains an INITV_IDX. The end of a link is specified by a zero INITV_IDX. Aggregate values are grouped into a separate links headed by the blkfield, which must not be the zero INITV_IDX.

kind : Kind of this INITV entry, see Table 2.30.

Table 2.30: INITVKIND

Name	Value	Description
INITVKIND_SYMOFF	1	value is the address of the symbol (st) plus offset (ofst)
INITVKIND_ZERO	2	integer value zero
INITVKIND_ONE	3	integer value one
INITVKIND_VAL	4	an integer, floating point, or string, specified by a TCON (tc)
INITVKIND_BLOCK	5	specifies another list or tree of INITVs
INITVKIND_PAD	6	amount of padding in bytes
INITVKIND_SYMDIFF	7	value is the difference of the addresses of a label and a symbol (lab1 - st2)
INITVKIND_SYMDIFF16	8	same as INITVKIND_SYMDIFF, except the value is 2 bytes in size
INITVKIND_LABEL	9	value is the address of the label (lab)

repeat1 : Specifies the repeat factor of the value in this INITV entry. This cuts down the number of unnecessary duplicates. A repeat factor of one means only one instance of the value is needed. The repeat factor is never zero, except for INITVKIND_ZERO, INITVKIND_ONE, and INITVKIND_VAL, which use repeat2 instead.

st/ofst : For INITVKIND_SYMOFF, the value of this entry is equal to the address of the symbol specified by st, plus the byte offset specified by ofst.

label : For INITVKIND_LABEL, the value of this entry is equal to the address of the label specified by lab.

lab1/st2 : For INITVKIND_SYMDIFF or INITVKIND_SYMDIFF16, the value of this entry is equal to the difference between the addresses of the label specified by lab1 and of the symbol specified by st2. It is a signed value equal to (lab1 - st2). For INITVKIND_SYMDIFF16, thesize of the value is 2 bytes.

mtype/repeat2 : For INITVKIND_ZERO and INITVKIND_ONE, this entry specifies an integral value of zero and one respectively. The WHIRL data type (signed/unsigned, size, etc.)is specified by mtype. It uses repeat2 as its repeat factor instead of repeat1.

tc/repeat2 : For INITVKIND_VAL, this specifies a TCON for the scalar constant value. It uses repeat2 as its repeat factor instead of repeat1.

pad : For INITKIND_PAD, this specifies the padding in bytes. The padded value is undefined.

2.13 BLK_TAB

Each entry of this table gives information about the layout of a data block, which corresponds to a contiguous chunk of memory in the user program. Program variables are laid out with respect to data blocks. This table is created by the back end and is usually local to the back end, but can be written out to the file. The BLK entry has the following structure, size 16 bytes:

Table 2.31: Layout of BLK

Offset	Field	Description	Field size

Table 2.31: Layout of BLK

Offset	Field	Description	Field size
byte 0	size	size of the block	2 words
byte 8	align	alignment of the blocks: 1, 2, 4, 8	2 bytes
byte 10	flags	flags for this field, see Table 2.32	2 bytes
byte 12	section_idx	section index (0 if not a section) • refers to the section info in data_layout.cxx	2 bytes
byte 14	scninfo_idx	Elf scninfo_idx (0 if not a section) • refers to the Elf section info in cgemit.cxx	2 bytes

Table 2.32: Miscellaneous Attributes of a BLK Entry

Flag	Value	Description
BLK_SECTION	0x0001	represents an Elf section
BLK_ROOT_BASE	0x0002	block should not be merged
BLK_IS_BASEREG	0x004	block that maps into a register
BLK_DECREMENT	0x0008	grow block by decrementing
BLK_EXEC	0x0010	executable instructions (SHF_EXEC)
BLK_NOBITS	0x0020	occupies no space in file (SHT_NOBITS)
BLK_MERGE	0x0040	merge duplicates in linker (SHF_MERGE)
BLK_COMPILER_LAYOUT	0x0080	layout of all symbols within this block is done by the compiler • this implies that user's code cannot legally use address arithmetic to move from one of the symbols to another

2.14 STR_TAB

This table holds all character strings for names of symbols, types, labels, etc. This table can be viewed as a block of storage area for character strings. STR_IDX is the index to this table, and is actually an offset in this block of storage; it gives the byte offset of the starting character of a literal string. All strings are null-terminated, and the first character of the block is always null. Thus, a zero STR_IDX represents a null string. Wide characters or unicode for names are not yet supported.

2.15 TCON_STR_TAB

This table holds all character strings defined in the user program. It is very similar to STR_TAB, with the exception that the strings need not be null-terminated, and null characters are allowed anywhere within the string. The exact length of each string is explicitly specified.

2.16 LABEL_TAB

Each entry of this table is a LABEL, which gives the information associated with a WHIRL label. The index to this table is the WHIRL label number.

The LABEL entry has the following structure:

Table 2.33: Layout of LABEL

Offset	Field	Description	Field size
byte 0	name_idx	STR_IDX to the name string, must be zero if no name	1 word
byte 4	flags	LABEL flags	3 bytes
byte 7	kind	kind of label	1 byte

Table 2.34: LABEL

Kind Name	Value	Description
LKIND_DEFAULT	0	ordinary label
LKIND_ASSIGNED	1	
LKIND_BEGIN_EH_RANGE	2	
LKIND_END_EH_RANGE	3	
LKIND_BEGIN_HANDLER	4	
LKIND_END_HANDLER	5	

Table 2.35: Miscellaneous Attributes of an LABEL Entry

Flag	Value	Description
LABEL_TARGET_OF_GOTO_OUTER_BLOCK	0x000001	control might be passed from outside of the current block to this label.
LABEL_ADDR_SAVED	0x000002	address of this label is saved to a variable
LABEL_ADDR_PASSED	0x000040	address of this label is passed to another PU as actual parameter PREG_TAB

2.17 PREG_TAB

Each entry of this table is a PREG, which gives the information associated with a pseudo-register in WHIRL. Pseudo-register numbers 0–71 are reserved for dedicated hardware pseudo-registers. All compiler-generated pseudo-registers start with number 72. As a result, the index to this table is the pseudo-register number, minus 71 (Note: by definition, index 0 to the PREG_TAB is reserved for undefined value). The PREG entry has the following structure:

Table 2.36: Layout of PREG

Offset	Field	Description	Field size
byte 0	name_idx	STR_IDX to the name string, must be zero if no name	1 word

2.18 ST_ATTR_TAB

Each entry of this table associates certain attribute with an ST entry. Symbol attributes specified here usually cannot be represented by a single bit, and are possessed by a very small subset of the ST entries, and thus are too expensive to be included as part of the ST entry proper. For most PU, this table is expected to be empty.

The ST_ATTR entry has the following structure, size 12 bytes:

Table 2.37: Layout of ST_ATTR

Offset	Field	Description	Field size
byte 0	st_idx	ST_IDX of the corresponding symbol	1 word
byte 4	kind	kind of the ST_ATTR, see Table 2.38	1 word
byte 8	reg_id	dedicated (physical) register associated with this symbol • symbol must have ST_ASSIGNED_TO_DEDICATED_PREG bit set	1 word
byte 8	section_name	STR_IDX of the name of the Elf section where this symbol is defined • symbol must be in global scope	1 word

Table 2.38: Kinds of ST_ATTR

Name	Value	Description
ST_ATTR_DEDICATED_REGISTER	0	dedicated register
ST_ATTR_SECTION_NAME	1	section name

2.19 FILE_INFO

This structure is not really part of the symbol table, it holds miscellaneous information that is derived from the symbol table but does not fit well in any global symbol table. Typically, this information is needed by the compiler backend to set up proper mode of operation before any PU is processed.

A FILE_INFO has the following structure, size 8 bytes:

Table 2.39: Layout of FILE_INFO

Offset	Field	Description	Field size
byte 0	flags	misc. attributes, see Table 2.40	1 word
byte 4	gp_group	gp-group id of this file, 0 for single GOT file	1 byte
byte 5	unused	unused, must be zero	3 bytes

2.20 Backend-Specific Tables

This section describes addition symbol tables that are created and used solely by the compiler backend. Each entry in these tables holds addition information associated with the corresponding regular symbol table entries. They are discarded at the end of the backend's processing and are never written out to a

Table 2.40: Miscellaneous Attributes of FILE_INFO

Flag	Value	Description
FI_IPA	0x00000001	IPA generated file
FI_NEEDS_LNO	0x00000002	some PUs in this file has the flag PU_MP_NEEDS_LNO set
FI_HAS_INLINES	0x00000004	some PUs in this file has the flag PU_HAS_INLINES set
FI_HAS_MP	0x00000008	some PUs in this file has the flag PU_HAS_MP set

file. Note that these tables are not part of the WHIRL symbol table specification and are implementation specific. The following descriptions apply only to the current implementation of the Open64 compiler.

2.20.1 BE_ST_TAB

This table is parallel to the ST_TAB. Each entry of this table is a BE_ST, which corresponds to an ST entry. The same ST_IDX is used to index an BE_ST entry in a BE_ST_TAB and the corresponding ST entry in the ST_TAB.

The BE_ST entry has the following structure, size 8 bytes:

Table 2.41: Layout of BE_ST

Offset	Field	Description	Field size
byte 0	flags	BE_ST flags	1 word
byte 4	io_auxst	pointer to an internal data structure used by the Fortran I/O routines.	1 word

Table 2.42: Miscellaneous Attributes of an BE_ST entry

Flag	Value	Description
BE_ST_ADDR_USED_LOCALLY	0x00000001	address of this symbol is taken somewhere within the current PU • this flag is computed based on the backend's analysis
BE_ST_ADDR_PASSED	0x00000002	address if this symbol is passed by reference • this flag is computed based on the backend's analysis • this flag is different from ST_ADDR_PASSED, which is set by the frontend based on the source language's semantics
BE_ST_W2FC_REFERENCED	0x00000004	whirl2c or whirl2f sees a reference to this symbol
BE_ST_UNKNOWN_CONST	0x00000008	symbol is a constant but with unknown value • generated by LNO
BE_ST_PU_HAS_VALID_ADDR_FLAGS	0x00000010	indicate that the BE_ST_ADDR_USED_LOCALLY and BE_ST_ADDR_PASSED bits are valid for the PU specified by corresponding ST entry. • valid only for CLASS_FUNC • depending on the optimization level, the above two BE_ST_ADDR flags might not be valid • tail-call optimization can be performed only when BE_ST_PU_HAS_VALID_ADDR_FLAGS is set

Table 2.42: Miscellaneous Attributes of an BE_ST entry

Flag	Value	Description
BE_ST_PU_NEEDS_ADDR_FLAG_ADJUST	0x00000020	indicate that the ST_ADDR_SAVED and ST_ADDR_PASSED bits are no longer valid • typically set by the MP-lowerer • needs to recompute the above two bits before moving on the next phase in the backend

2.21 Symbol Table Interfaces

The symbol table interfaces are described in a separate document. An online version can be found in http://sahara.mti.sgi.com/Projects/Symtab/porting.html/.

2.22 Symbol Table Programming Primer

(Based on: Mike Fagan and Nathan Tallent. "Design and Implementation of whirl2xaif and xaif2whirl." Rice Technical Report, TR03-16, 2003.)

WHIRL's symbol table can be difficult to work with. What follows are some important things we have learned about it.

First, it is implemented in C++ (templates and classes) with a C function wrappers for an interface! Most likely (or one hopes!) this has something to do with code reuse, where the symbol table implementation was rewritten but its interface was preserved.

Secondly, WHIRL's scoped symbol table works as advertised for intra-procedural operations. However, once one desires to examine symbol tables in an inter-procedural fashion, the full moon rises and the werewolf starts to howl. Or at least it seems that way.

Most significantly in this connection, it is important to realize that a WHIRL PU – program unit, representing a procedure or function – is *not* a self-contained encapsulation. Instead, it is a wrapper for a WHIRL tree and some PU specific symbol tables. However, nodes in the WHIRL tree contain symbol table references that do *not* point directly into these tables, but refer to tables within the global Scope_tab[] (Scope Table), the table of the *current* visible lexical scopes. Consequently, while multiple WHIRL trees and symbol table can reside in memory, the only way to access the symbols for a PU is when it is within the *current* lexical scope. Hence all the symbol table references in other WHIRL trees effectively point to junk. The global 'current' pointers must be updated *each time* one moves to a different PU. Because Open64 did not provide a good way of switching between PUs during inter-procedural algorithms, we developed a tolerable way of doing so.

Thirdly, types in the WHIRL symbol table are sometimes difficult to keep straight. Here are the most important types:

SYMTAB Not actually a type, but refers to all of the tables at a particular level/scope. Besides a global scope, there is a local scope for each nested PU. Each scope contains a number of different tables, some of which are common to all levels (e.g. ST_TAB) and some of which are specific to global (e.g. PU_TAB) or local levels (e.g. LABEL_TAB).

SYMTAB_IDX The type of an index into the scope table Scope_tab[]. The global scope is always at the index GLOBAL_SYMTAB; the scope for the current lexical PU is at index CURRENT_SYMTAB. (This is set by the WHIRL reader function Read_Local_Info().)

ST_TAB The type of the symbol table proper, a table that appears at all lexical levels.

ST_IDX A two-part index into any ST_TAB within the Scope_tab[]. The two-part bit field contains an index into the ST_TAB at a certain lexical level.

ST The type of a ST_TAB entry.

And here are the global data that actually implement the symbol table:

Scope_tab[ ] An array of SCOPEs, indexed by SYMTAB_IDX, the lexical level. A SCOPE contains pointers to all the tables for a lexical level, including a ST_TAB.

St_Table[ ] Essentially a class wrapper for Scope_tab[] with member functions for indexing both the Scope_tab[] and the appropriate ST_TAB with a ST_IDX. (TABLE_INDEXED_BY_LEVEL8_AND_INDEX24)

Chapter 3

Appendix

List of Figures

Index