For: rft
Printed on: Tue, Feb 2, 1993 09:18:07
From book: asm-doc
Document: spec
Last saved on: Tue, Feb 2, 1993 09:17:11

1. Introduction

1.1. History

This document describes the assembly language inline feature implemented in the C compiler provided by HCR for iWarp. The version defined here is similar to that provided in the AT&T System V.3 PCC2 compilers for the 3b2 and Intel 386 processors. Steve McGeady provided the initial description and also provided valuable consultation during the development of this definition. The present implementation is a result of the mutual efforts of Lorne Wilkinson and Paul Jackson of HCR, and Herb Mayer and Roy Touzeau of Intel.

1.2. Overview

The assembly code inline mechanism ("asm–inline") provides a flexible way of inserting specific assembly language code sequences into the assembly output generated by the C compiler. In contrast to the conventional *asm* feature found in most C compilers, the asm–inline allows the user to write assembly language sequences without knowing the details of the compiler implementation, such as what specific machine registers must be used in the assembly code. He may write a sequence of assembly language instructions using "variable names" in the place of the register names in the instructions. When the compiler inlines the assembly language instructions it will replace the "variable names" with appropriate register assignments.

In many ways, the inline appears like a function call, but it results in very efficient code by avoiding the overhead of an actual function call. The user defines and uses an asm–inline much the same way he would define and use a static C function. The inline has a name and parameters, and is "invoked" by using its name in a C expression. The inline definition specifies formal parameters, local entities and assembly code sequences. Alternate code sequences may be defined within the inline definition. When the compiler expands the insert, a matching and selection process, described in section 3., chooses one of the code sequences to inline. The matching process provides a mechanism for the asm–insert to be sensitive to the register environment the compiler has produced for a particular invocation of the inline. This permits the compiler to generate a very efficient integration of the expanded assembly code into the existing code stream. The implementation avoids the normal overhead associated with function calls by treating the inline code sequence as a template for a complex operator. While this allows very efficient code generation, it necessarily puts special restrictions on the user in writing the asm–insert. These restrictions will be specified later in this document.

The matching and leaf selection process may appear confusing and complex at first, but it is important to understand it because it is the means by which the compiler efficiently integrates the user supplied assembly code with the surrounding compiler generated code. The process is complicated by the fact that the compiler will generated different kinds of code for the same source at different optimization levels. The writer of the asm–inline must keep this in mind, otherwise the insert may work at one optimization level and cause compile–time errors at another. In this case, this is not a compiler failure but indicates that more careful coding of the insert is required.

A number of examples are given in section 7. and are correlated to the early expository sections of this document. These supply supplemental reading alongside the text and may be helpful.

1.3. Definitions

actual argument — This term refers to arguments passed to an assembly language inclusion function upon specific invocation of that function by the caller.

actual call — This term refers to the the generation of a conventional call to a function with the name of the assembly language inclusion function. (That is, expansion of the insert does *not* take place.)

aggregate type — is a structure, union or array type. However, since C converts references to an array in the argument list of a call to a pointer to an array and does not allow arrays return values, only structures and unions are relevant aggregate types in the context of the asm–inline.

argument registers — These are registers utilized by the compiler for storing the arguments for a function call. For iWarp, registers *ar0* through *ar15* are the argument registers.

assembly body — This term refers to the actual iWarp assembly language instructions which will be subjected to expansion and result in the final assembly code sequence when the compilation process is complete.

assembly language inclusion function — This term refers to the construct for assembly language inlining, described in this document. It is similar to a C function, except that the body contains control lines and iWarp assembly language instructions.

asm–inline — This term is a synonym for ‘assembly language inclusion function’.

asm–insert — This term is a synonym for ‘assembly language inclusion function’.

caller — This term refers to the function which calls an assembly language inclusion function.

complete match — This term refers to a condition where all the matching specifications in a particular control line result in a successful match.

control line — The control line is a line in the asm–inline function which begins with a percent (%) and controls selection of code sequences for inclusion.

control space registers — These registers are the control and status registers that are not directly accessible via the register file, and have direct and immediate effects upon the hardware. These registers are not known by the compiler.

formal parameter — This term refers to the name defined in an insert that will be bound at compile time to the corresponding actual argument.

function declaration — See section 4.5, page 76 of “The C Programming Language”, by Kernighan and Ritchie.

function definition — See section 4.5, page 76 of “The C Programming Language”, by Kernighan and Ritchie.

function return registers — These are registers utilized by the compiler for the storage of intermediate values during the evaluation of expressions, in particular for the storage of values returned by functions. For iWarp, registers *fv0* through *fv3* are the function return registers.

insert — This term is a synonym for ‘assembly language inclusion function’.

leaf — The leaf section of an insert consists of a control line and an assembly body.

matching — This term refers to the process of comparing the specified storage classes of formal parameters and return value with the actual storage classes of the actual arguments and invoking expression.

matching specification — This term refers to the part of the control line which defines the matching criterion for a particular identifier.

permanent registers — Registers *pr0* through *pr55* are the permanent registers. At lower optimization levels, permanent registers are assigned by the compiler for variables declared by the user to be of *register* storage class and of a fundamental type. In addition, PCO may allocate values to some of these registers. At higher optimization levels the distinction between permanent and scratch registers is irrelevant to asm–inlining because the compiler internally uses virtual registers and does not make assignments to actual registers until after asm–insert processing is done.

pseudo call — This term refers to the C statement that uses an assembly language inclusion function.

scratch registers — Registers *sr0* through *sr15* are scratch registers. These are registers used by the compiler for the storage of intermediate values during the evaluation of expressions. At higher optimization levels the distinction between permanent and scratch registers is irrelevant to asm–inlining because the compiler internally uses virtual registers and does not make assignments to actual registers until after asm–insert processing is done.

storage class — This is the property of an object (actual parameter, formal parameter or local variable) which defines the type of storage it has (permanent register, scratch register, or literal constant). The storage class of an object is used in the control line for matching actual and formal parameters as well as defining the local variables.

user hardware registers — These registers are *ev0* through *ev7*, *sc0* through *sc15*, *lm0* through *lm5* and *ga0* through *ga3*. These registers are special iWarp hardware registers which are accessible in the register file. The compiler does have knowledge of these registers, but they are only used for special intrinsic functions in software pipelining inner loops.

1.4. Syntax notation

When describing the syntax of the asm–inline, symbols printed in normal (Times) bold typeface are terminal symbols and must be used exactly as typed. Non–terminal symbols, which refer to other grammar rules, are printed in *italic* typeface. Syntax rules begin with a non–terminal symbol followed by a colon (:). Alternative rules are listed on successive lines. For example,

word :
  letter_list
  letter

letter :
  A
  B

The first rule specifies that a "word" consists of a single *letter* or a single *letter_list*. The second rule specifies that *letter* may be the letter "A" or the letter "B".

Optional symbols (terminal or non–terminal) are subscripted with *opt* as in:

letter_list :
  letter_list $,_{opt}$ *letter*

The comma may optionally appear in the list of letters, as in "AAB,BBAAB".

2. C language interface

2.1. Function definition

The assembly insert has the overall form of a sequence of assembly language instructions in a C–like function "wrapper". An asm–inline function must have a storage class of **asm**. The informal grammar below gives the form of the wrapper.

asm_function_definition :
  asm *type_specifier*$_{opt}$ *asm_function_declarator* *asm_function_body*

asm_function_declarator :
  asm_declarator **(** *asm_param_list*$_{opt}$ **)**

asm_param_list :
  formal_param_id
  asm_param_list **,** *formal_param_id*

formal_param_id :
  identifier

asm_declarator :
  identifier
  ***** *identifier*

a*sm_function_body* :
  { *asm_block* **}**

So, the asm_inline has the superficial appearance of a standard C function, such as:

```
asm int foo(x) {  ...  }
```

where ... represents the *asm_block* described in section 3.

The *type_specifier* is optional and the default type is **int**. The asm–inline may not be given an aggregate type.

Unlike standard C functions, formal parameters do not need declarations.

Formal parameter names must follow the lexical conventions for C identifiers. The asm–inline reserved words[1], and assembly language reserved words must not be used as names for parameters or local variables. Since the expansion mechanism is a straight substitution mechanism, using names that are assembler mnemonics will almost certainly result in illegal assembler code. Identifiers used in the insert are not checked for collision with C reserved words or assembler reserved words.

Because the asm–inline mechanism gives the user access to the machine level storage facilities (i.e. registers), it is not possible to classify the parameter passing behavior as strictly "by–value" or

1. Reserved words for the asm–line are those which appear in normal bold type in the syntax descriptions, including all of the storage class names.

“by_reference”. Neither behavior can be enforced. Nevertheless, to avoid the problems described in section 5. the user must not attempt to use parameters to pass information back to the calling routine.

The asm–inline definition may appear anywhere a standard C function definition may appear, except that the definition must appear before the first use. If a reference is made to an asm–inline function lexically before its definition, an actual call will be compiled, but a fatal compile time error will be issued when the compiler later finds the definition.

2.2. Function invocation

The asm–inline is invoked as a standard C function, such as:

```
x = asm_foo(y);
```

where asm_foo has been previously defined as an asm–inline function.

An asm–inline can also be used in a more complex expressions, such as:

```
x = asm_foo(asm_foo2(y)+b) & (asm_foo(z) << 2);
```

Note: An asm–inline function call may not be used anywhere a value with an aggregate type is required.

The order of evaluation of the actual arguments to the asm–inline is the same as that for an actual call because when the insert is expanded it may be converted into an actual call. Since the C language does not specify the order of evaluation of arguments to calls (i.e., it is implementation dependent), the order of evaluation of arguments to an asm–inline is also unspecified.

An actual argument of type **enum** will be treated as an **int**. Actual arguments of type **char** and **short**, or **unsigned** versions thereof, will be subject to the standard C semantics for promotion. However, to simplify the use of asm–inserts the normal conversion of floating point values and return values is suspended. See the **iwcc** man page for information on the “–fnoasmwiden” switch which is used to control the C compiler’s handling of conversions in the context of an asm–inline.

An actual argument which is a string or array name will be treated according to normal C semantics. In particular, the string will be stored in a compiler generated location and the address of the string will be passed as a literal constant. Similarly, in the case of using the array name as an actual argument, the address of the array will be passed as a literal constant.

3. The Asm function body

This section describes in detail the syntax and semantics of the insert itself. Section 3.1. provides an overall description of the whole insert and the subsequent sections describe each part in detail.

3.1. Description

The asm–inline body is made up of zero or more *leaves*, where each leaf consists of a control line followed by zero or more assembly language instructions. (Note that the control line is not an optional element of a leaf.) The leaf is terminated by the next control line or by the end of the *asm_block* (indicated by the closing brace). The control line begins with a percent (**%**) and terminates with a semi–colon and new–line (**\n**). A control line may be continued on the next physical line by escaping the new–line. The control line defines entities used by the leaf and provides control information for conditional expansion of the leaf, as described in section 3.2. Exactly one leaf will be expanded to provide the assembly code.

Comments may be inserted into the asm–inline body by using either the C or the assembly language format. A C style comment will not be passed on through to the assembler, whereas an assembly language style comment (a line beginning with "– –") will be passed on to the assembly code. A C style comment may appear anywhere in the *asm_block*, but an assembly style comment may only appear in the *expansion_list* (e.g., it may not appear in a control line). **Note:** assembly language style comments are scanned by the C preprocessor; consequently, any macros (intended or otherwise) will be expanded.

The syntax of the *asm_block* is given by:

asm_block :
  $asm\_leaf\_list_{opt}$

asm_leaf_list :
  asm_leaf
  asm_leaf_list *asm_leaf*

asm_leaf :
  % *control_line* **;** **\n** *expansion_list*

control_line :
  control_list
  control_line **** **\n** *control_list*

expansion_list :
  expansion_line
  expansion_list *expansion_line*

expansion_line :
  text operand_list
  label :
  arbitrary_string

operand_list :
  operand
  operand_list **,** *operand*

operand :
formal_param_id
local_var_id
return

For example, the following asm–inline has one leaf which is always expanded when the asm–inline is invoked. The resulting assembler code will set the control space register, *utimer0*, to zero.

```
asm void foo()
{ % tmpreg x;
        ldlitlz 0, x
        movecsr x, utimer0
}
main()
{
        foo();
}
```

Overall, the asm–inline definition has the form of a C function definition with the storage class **asm**. The declaration **tmpreg x** prompts the compiler to allocate a register for the symbol **x**. All the lines that follow the control line up to the end of the function are passed directly to the assembly code output from the compiler after declared symbols have been replaced by appropriate register names. The next section discusses the syntax and semantics of the control line and its influence on the generation of the assembly code.

If the asm–inline is to return a result value to the calling expression, the control line must contain a specification for the special local variable, "return", and the body must contain a definition of the variable "return" (e.g., a movereg with "return" as a target).

Note: the assembly code supplied in a leaf is passed without being interpreted or checked (except as described in section 3.2.) to the assembler. It is the programmer's responsibility to make sure that the code is correct and does not interfere with the compiler's normal register assignments. This topic is discussed further in section 5.

3.2. Control Line

The control line provides four important functions:

1. it specifies the storage class of each formal parameter and of the return value for the asm–inline,
2. it defines identifiers used as local variables within the asm–inline leaf associated with the control line,
3. it defines labels used within the leaf associated with the asm–inline leaf, and
4. it specifies certain actions to be taken either in conjunction with expansion of the leaf or instead of expansion.

In conjunction with (1), the control line provides for conditional expansion of the leaf based on the process of matching. The syntax of the control line is given next, followed by details of these functions.

control_line :
control_list
control_line **\ \n** *control_list*

control_list :
control
control_list *control*

control :
arg_spec **;**
use *reg_list* **;**
lab *label_list* **;**
error "*string*" **;**
call ;
call *identifier* **;**

arg_spec :
storage_class name_list

name_list :
name
name_list **,** *name*

storage_class :
(see text)

name :
formal_param_id
local_var_id
return

label_list :
label_id
label_list **,** *label_id*

For a complete syntactic description of *storage_class*, see the appendix. Here the syntax of *storage_class* is described in a concise way.

A storage class may be one of the following:

register
tmpreg
regnum
const
csregnum

In addition, *register* and *tmpreg* may have a prefix of *d* (for *double*). Also, *regnum*, *const*, and *csregnum* may have one of the following suffixes:

(*number*)
(*number* : *number*)

where *number* may be either a positive or negative decimal integer or hexadecimal number.

3.2.1. Scanning the control line

All the control entities except *arg_spec* (argument specification) are called non–matching control directives because they are never included in the matching process described below. The argument specification (*arg_spec*) includes the declaration of arguments (*formal_param_id*), local entities (*local_var_id*) and the optional return specification. The return specification does not affect matching, but is required except when the asm–insert is declared to be of type "void". The name **return** is the reserved name of a special local entity that must be given the storage class of tmpreg or dtmpreg. The compiler allocates the function return register(s) to this variable. So, it can be used to provide a result value back to the calling routine.

If the asm–inline has *n* formal parameters, then each control line must have argument specifications declaring at least *n* entities (except as described below in section 3.2.4.). The entities declared in the argument specifications are associated lexically left–to–right with the formal parameters. If any control line has more than *n* entities declared in it's argument specifications then only the first *n* are considered when attempting to find a complete match. Section 3.2.2. describes the matching criteria for individual declarations. Any entities left over after pairing the first *n* of them with parameters are definitions of variables local to the leaf. The storage class used with these left–over entities must be either 'tmpreg' or 'dtmpreg'. If they are not of one of these classes, a fatal error will occur. If one or more local variables are defined the compiler will allocate them registers or register pairs. A variable with a storage class of 'tmpreg' will be allocated a single register and a variable with a storage class of 'dtmpreg' will be allocated a register pair. Non–matching control directives are discussed in section 3.2.4.

Consider the following example. Suppose an asm–inline function has three formal parameters and the following control line:

```
% const x; tmpreg y,z;
```

Since there are three formal parameters the storage classes shown pertain to these parameters. So, no local variables would be allocated. The leaf associated with this control line will be expanded if the asm–inline function is invoked with a call which has a literal integer constant as the first actual argument and single precision variables as the second and third actual arguments. (Note that this works because the C compiler suspends the normal promotion of arguments for asm–inserts. This behavior is governed by the "–fnoasmwiden" command line option.)

If, on the other hand, the asm–inline function had two formal parameters, then "z" would be a local variable name and the compiler would allocate a (single) register for this entity.

Details of the matching process are given in sections 3.2.2. and 3.2.3.

3.2.2. Storage classes and their matching characteristics

The following table shows all the storage classes which can be used to match actual arguments and specifies what kind of actual argument will cause a match. It is important to realize that the matching process takes place within the context of the compiled code. This is why, for example, the "const" storage class will match an *address*. This could occur, for example, if the actual argument in the source code is an array reference to a static array.

storage class:	**matches if Actual Argument is:**
const	any integer constant (including a literal address).
const(n)	integer constant of value 'n'; 'n' is a signed integer or hexadecimal value.
const(n:m)	integer constant in inclusive range from n to m; 'n' and 'm' are signed integer or hexadecimal values.

regnum	any register–file register number. Actual argument must be a literal in the range of the current register–file register numbers.[1]
regnum(n)	a particular register–file register number ‘n’, ‘n’ is a hexadecimal or a signed integer value. The actual argument and ‘n’ must be in the range of the current register–file register numbers.
regnum(n:m)	any register–file register number in the inclusive range from n to m; ‘n’ and ‘m’ are hexadecimal or signed integer values. The actual argument, ‘n’, and ‘m’ must all be in the range of the current register file register numbers.
csregnum	any control space register number. The actual argument must be a literal in the range of the current control space register numbers.
csregnum(n)	the control space register number ‘n’; ‘n’ is a hexadecimal or a signed integer value. The actual argument and ‘n’ must all be in the range of the current control space register numbers.
csregnum(n:m)	any control space register number in inclusive range from n to m; ‘n’ and ‘m’ are hexadecimal or signed integer values. The actual argument, ‘n’, and ‘m’ must all be in the range of the current control space register numbers.
register	any iWarp single register (either permanent or scratch).
dregister	any iWarp double register (either permanent or scratch).
tmpreg	any iWarp single scratch register.
dtmpreg	any iWarp double scratch register.

Note: since the ‘register’ storage class matches both permanent and scratch registers, if both the ‘register’ and ‘tmpreg’ storage classes are used in the same insert, the control line containing the ‘tmpreg’ storage class must preceed any control which uses the ‘register’ storage class for the corresponding parameter. Otherwise, the leaf having the ‘tmpreg’ specification can never be expanded.

Note: At higher optimization levels all registers are “virtual”; there is no distinction between “permanent” and “scratch” registers.

Note: within the asm–inline “register” denotes a read–only register; “tmpreg” denotes a read–write register. The compiler’s register allocator uses this information. If the user writes to a register that has been declared “register” he will endanger the integrity of his computation.

Note: floating point constants are *not* matched by the ‘const’ storage class. They can only be matched by ‘dtmpreg’ by default or by ‘tmpreg’ if the “–fsingle” switch is specified to the compiler. This characteristic is a result of the fact that the C compiler stores all floating point constants in compiler generated temporaries. Consequently, any reference to a floating point constant is compiled as a reference to a compiler generated double precision “variable”.

Note: The system include file <regnum.h> is provided to define register names to the correct register numbers. This file must be used to ensure that the correct register numbers are used across releases of the system software. (See the examples for this section.)

Note: ‘const’ ***is not interchangable*** with ‘csregnum’ or ‘regnum’ eventhough they all match literal constants. If the ‘const’ storage class is used where a ‘csregnum’ or ‘regnum’ storage classes should be used then the proper control space or register file register *will not* be properly substituted.

1. The standard iWARP include file <regnum.h> defines mnemonics for the current register numbers.

3.2.3. Matching semantics

Argument matching is a two pass process. The first pass considers the storage class of each actual argument exactly as it exists in the expression of the call. The control lines of the insert are searched in the order in which they appear in the asm–inline body. The matching algorithm seeks a complete match by comparing the storage class of each parameter with its corresponding argument specification. If all comparisons provide a match then a complete match has been found. The first leaf that is found whose control line provides a complete match will be expanded.

If none of the control lines provided a complete match during the first pass, then a second pass over the control lines is performed. During the second pass, each actual argument is assumed to match either the 'tmpreg' or 'dtmpreg' storage class, depending on the data type of the actual argument. For the first leaf that a complete match is found, based on this assumption, the assembly language inclusion mechanism allocates the required registers and generates the additional prologue code to perform the coercion of the actual arguments to 'tmpreg' or 'dtmpreg' as appropriate before expanding the leaf.[1]

If no control line is specified in a leaf, or an empty control line is found, or no match is found after the second pass and the exceptions defined in section 3.2.4. do not apply, then a fatal error occurs and code generation for the entire module is abandoned.

3.2.4. Non–matching control line directives

If a leaf is found for which a match occurs, as defined in section 3.2.3., and the control line contains either or both of the directives "lab" and "use", then the assembly body is expanded and the actions described below are taken.

lab	Declares one or more identifiers as labels in the insert. A label declared in an insert must have a unique definition point. When the insert is expanded, the compiler allocates a label which is unique in the assembly output for the compilation unit.
use	Informs the compiler that certain registers are used in the following assembly body. This information is required by the compiler's register allocator when the user codes specific register–file register names into his asm–insert.

The actions described below are taken without the expansion of the assembly body, even if one is specified, under the following conditions:

1. a leaf is found for which a match occurs, as defined in section 3.2.3., and the control line contains one of the following directives,
2. *no* leaf is found for which a match occurs and one of the control lines in the body contains exactly one of the following directives.

error "msg"	Compile time error is generated and the optional message "msg" is output. No code is generated and in–line expansion is abandoned. This will only occur after coercion of all formal parameters in the control lines has failed to result in a match (i.e. after the second pass).
call	An actual call to an external function with the same name as the insert is generated. This directive is processed during the second pass.

1. In this way a control line can be written to always match single or double precision parameters, regardless of the actual arguments. This is an important feature for writing general purpose asm–inline functions.

call name — An actual call to an external function with the name ‘name’ is generated. This directive is processed during the second pass.

A single control line may contain only one these last three control line directives. If one of the last three control directives is used, it usually should be placed by itself on the last line of the insert (See the examples for this section). **Note** that since a “call” directive is always matched during the second pass, judicious placement can be used to distinguish between leaves that are to go through the coercion process and those that are not to go through coercion. The leaves preceeding the “call” will go through the coercion process and those following it will not.

3.3. Binding actual to formal parameters

When a complete match is found the “values” of the actual parameters are bound to the formal parameters. But since the asm–inline deals with the storage entities associated with the actual arguments instead of the values stored in the actual argument as does a normal C function, the “value” bound is quite different from that bound in a normal C function call. In the case of the asm–inline, the value of the actual argument is the object which was considered in the matching process (See section 3.2.2.). For example, the storage class ‘const’ matches a literal constant, thus the value bound to the corresponding formal parameter is the literal value matched. One might think that this would also be the case with strings; however, C semantics defines that a string is actually converted to a pointer to a compiler generated location where the string is stored. So, the asm–inline would see an actual argument which is a constant but whose value is an address. (The examples in section 7.4. illustrate this.) In the case of a storage class of ‘tmpreg’ or ‘register’, the actual value matched is the name of the register holding the value of the variable used as the actual argument. So, the formal parameter will be bound to the name of the register holding the value of the variable which is used as the actual argument. Note that when the actual argument is a floating point constant it is not matched by the ‘const’ storage class but rather the ‘register’ or ‘tmpreg’ storage class. In this case, the value bound to the formal parameter is a register number (See the notes at the end of section 3.2.2.).

3.4. Limits on arguments and local variables

If the evaluation of any single argument results in an argument greater than a double word, then a fatal error will occur, and code generation for the entire module will be abandoned.

4. Expansion semantics

During expansion, all text in the selected leaf except instruction operands and names declared “lab” are copied without interpretation to the assembly output file. Any operand which is textually identical to a formal parameter is replaced with the value that was bound to the actual argument during the matching process (Section 3.3.). Any operand which is textually identical to a local variable is replaced with the register name assigned by the compiler to that local variable. Any operand which does not textually match a formal parameter, a local variable or “return” is copied to the assembly output file without interpretation.

Names that were declared as labels are replaced by actual label names generated by the compiler and guaranteed to be unique.

Within the asm–insert “register” denotes a read–only register; “tmpreg” denotes a read–write register. The compiler’s register allocator uses this information. If the user writes to a register that has been declared “register” he will endanger the integrity of his computation. The compiler checks many such conditions and issues an error message if they are violated.

Use of the ‘regnum’ and ‘csregnum’ storage classes requires special comment. If the storage class of the matched parameter is of the ‘regnum’ type, a reference to the register of that ordinal value is generated. For instance, if the actual argument is a constant with the value 508 (0x1fc),[1] reference to that parameter in the assembly body will be substituted with ‘ga3’. This register will not be saved before expansion and restored afterwards; users must exercise caution if referencing registers with this mechanism. If the storage class of the matched parameter is ‘csregnum’, a reference to the control space register of that ordinal value is generated. For instance, if the actual argument is a constant of value 1024 (0x400), reference to that parameter in the assembly body will be substituted with ‘fpaflg’.

1. The numbers given here are samples and may change from one release of the system software to another. The system include file <regnums.h> is provided which defines mnemonics for the current register numbers.

5. Optimizer interactions

5.1. Global optimizer interactions

The optimizer must assume that the assembly language inclusion mechanism obeys C semantics for function calls and arguments to function calls. Therefore, the user must be cautious when writing inserts. If he obeys the same rules as those for standard C function calls and arguments, the impact on the effectiveness of optimization will be minimal and program correctness will be preserved. The user could violate these rules by writing to a parameter which has been matched with a storage class of 'register'. Expansion of an insert with a 'register' storage class parameter will result in an effective pass by reference as opposed to a pass by value. If the formal parameter is written to in the assembly body, the value will also change for the caller function. This contradicts the C language call by value semantics. Since the optimizer assumes that C semantics are being followed (i.e. a parameter value cannot change) unpredictable results can occur. The optimizer is not permitted to make optimizations that result in code motion affecting the location of the use of an insert, since in this case the functionality of the insert could be compromised. Another problem is the fact that the optimizer could change the storage class of an argument to an insert, resulting in different behavior from the insert.

Given the insert:

```
asm int foo(a)
{
        % const a;
        ...
        % error;
}
```

Compilation of:

```
main()
{
        int i;

        i = 1;
        foo(1);
}
```

without optimization would complete as expected. However, compilation in the presence of optimization may result in the assembly language inclusion mechanism expanding the insert as if the user had written:

```
main()
{
        register int i;

        i = 1;
        foo(i);
}
```

because the optimizer has promoted *i* to a *register* storage class. This would result in a compile time error.

One solution to this and similar problems is to provide leaves for efficient implementation first (using ‘const’ and ‘tmpreg’ storage classes) followed by a catch–all leaf using the ‘register’ storage class definition.

6. User restrictions

The following rules summarize the restrictions that users must follow if they are to avoid writing inserts which will run erratically or will produce incorrect results.

1. Never modify any parameter unless it was declared with a ‘tmpreg’ storage class.
2. Never modify any entity on the stack.
3. Never modify a global entity.
4. Never allow a net effect on the stack pointer (SP).
5. Never branch from inside an insert to a label outside the insert or from one leaf to another.
6. Never write timing or sequence dependent code in an insert without using the **.begincrit** and **.endcrit** psuedo–ops to bracket the sensitive code.
7. Never call a routine directly; always use the **%call** control line directive.
8. Asm–insert definitions which include “syscall” must contain the following directives at the end of the definition:
```
%error <some msg.>;
%call;
```
9. Never use the **.flags** and **.noflags** assembler pseudo–ops. The compiler uses these and assumes they are not changed. Instead use the **flags** and **noflags** *prefixes* on the instructions for which this capability is needed.

The “Never modify” rules can be summarized by saying, “Only modify parameters and local variables which have been declared to have a ‘tmpreg’ storage class or specific hardware registers that the compiler does not know about”.

Rule 7. can be violated only if the user thoroughly understands and adheres to the call/return convention defined for his environment.

Because the optimizer may promote variables in a function to registers, it is recommended that the user always include a leaf with **%register** control directives for parameters in each insert. Otherwise, the insert may fail to find a match when the optimizer is used in compiling the program.

7. Examples

7.1. Examples for section 2.

The following example shows a simple inline definition which will always expand to perform an integer multiplication of its parameters and return the result. The invocation in the C program appears as a normal C function call. The inlining mechanism will expand the assembly code at the point of the pseudo–call.

```
asm int foo(x,y)
{ /* asm inline body */
        %  tmpreg x,y; tmpreg return;
        -- start of asm inline leaf
                fimul x,y,return
        -- asm inline end
}
main()
{       int a = 1,int b = 2,int c = 3;
        int x;

        x = foo(a,foo(b,c)) + 10;/* asm inline invocation */
        printf("%d\n",x);
}
```

7.2. Examples for section 3.1.

The following example shows a simple version of the insert shown above along with the expansion that results from an invocation. The ‘tmpreg’ storage class for x and y directs the inline mechanism to coerce the constants 1 and 2 into scratch registers when the expansion is done. If these constants were replaced by more complex expressions the actual in coming values would already be in scratch registers and the expansion mechanism would simply substitute those registers in the fimul instruction.

```
asm int foo(x,y)
{ /* asm inline body */
        % tmpreg x,y; tmpreg return;
        -- start of asm inline leaf
        fimul x,y,return
        -- asm inline end
}
main()
{       int x;

        x = foo(1,2);     /* asm inline invocation */
}
```

The following is the complete assembly listing from compilation of the program above. The insert is invoked on line 11 of the program. Actual inserted code is shown between the indented comments.

The “ldlit32” instructions were added by the inlining mechanism when it found the actual arguments to be constants and the storage class of formal parameters to be ‘tmpreg’. Also, the compiler adds .ai*xxxxx* nmemonics that the assembler can use to check the correctness of the insert code.

```
        .text
        .align  4
        .noflags
--      line 0, file ”e1.c”
        .file   ”e1.c”
        .version        ”3.0”
        .arch   ”3.2”
        .data
        .text
--      line 9, file ”e1.c”

        .align  4
        .globl  _main
_main:
.L49999:
--              skipped frame allocation due to register assigner hint
-- For main():
-- Prolog:

.L49998:
--      line 11, file ”e1.c”
        ldlit32 1, sr2
        ldlit32 2, sr1
        .beginai _foo
        .airegrw        sr0     -- RETURN: %tmpreg return
        .airegrw        sr1     -- %tmpreg y
        .airegrw        sr2     -- %tmpreg x
        .aibody
     -- start of asm inline leaf
     fimul sr2,sr1,sr0
     -- asm inline end

        .endai          -- foo
.L49997:
.L54:
        ret

        .endfunc
        .data
--      line -1, file ”e1.c”
```

7.3. Examples for section 3.2.

The following example shows the usefulness of the coercion mechanizm in writing simple, general asm–inlines. The first invocation of *foo* has an actual argument of type ‘const’ and the expansion provides a move to a register automatically. In the second invocation the actual argument is already in a register and the most efficient insertion is provided. No copies are necessary because the **x** is given storage class ‘register’ which guarantees that the register assigned to it will not be written.

```
asm void foo(x)
{
        % register x;
                movecsr x, utimer0
}
main()
{       register int x = 0;

        foo(0x3);
        foo(x);
}
```

The assembly output generated for this example follows (Assembly code generated by the compiler which is unrelated to the example has been deleted.):

```
--      line 7, file ”e4.c”
        ldlitlz 0,ar0   -- k=0x00000000(0)
--      line 9, file ”e4.c”
        ldlitlz 3,fv0   -- k=0x00000003(3)
        .beginai _foo
        .airegr fv0     -- %register x
        .aibody
                movecsr fv0, utimer0
        .endai          -- foo
--      line 10, file ”e4.c”
        .beginai _foo
        .airegr ar0     -- %register x
        .aibody
                movecsr ar0, utimer0
        .endai          -- foo
```

The line “ldlitlz 3,fvc” was inserted by the asm–inline expansion mechanism to coerce the constant 0x3 to the “register” storage class of **x**.

The following example contains two control lines and shows the expansion of alternate leaves based on the matching of constant and register storage classes. Also, the special scratch register *return* is allocated to return the value of the asm–inline to the calling expression. If the leaf with the ‘const’ storage class is provided because the *ld* instruction can take a literal constant in the offset position. This permits the expansion to be done efficiently in this case. Otherwise, the expansion would include a copy to a register similar the previous example.

```
asm int wierd_load(p,o)
{
  % register p; const o; tmpreg return;
      ld.w (p,o)+,return
  % register p; tmpreg o; tmpreg return;
      ld.w (p,o)+,return
}
fum(x)
int *x;
{ register int r=4;

 return(wierd_load(x,4)+wierd_load(x,r));
}
```

In the following output, some irrelevant assembly code at the beginning was deleted.

```
--      line 12, file ”e3.c”
        .beginai _wierd_load
        .airegrw        sr0     -- RETURN: %tmpreg return
        .airegr ar0     -- %register p
        .aibody
      ld.w (ar0,4)+,sr0

        .endai          -- wierd_load
        .beginai _wierd_load
        .airegrw        fv0     -- RETURN: %tmpreg return
        .airegr ar0     -- %register p
        .aibody
      ld.w (ar0,4)+,fv0

        .endai          -- wierd_load
        add     sr0,fv0
.L49997:
.L54:
        ret

        .endfunc
        .data
--      line -1, file ”e3.c”
```

The following example shows the use of the specific range limits available for certain storage class specifiers. Defines for the CS registers are in the system include file ‘regnums.h’.

```
#include <regnums.h>
asm setmemreg(reg,hi,lo)
{
        % csregnum(CSR_LMPROT) reg; const hi,lo; tmpreg z;
                ldlitl lo,z
                ldlith hi,z
                movecsr z,reg
        % csregnum(CSR_LMSIZE) reg; const hi,lo; tmpreg z;
                ldlitlz lo,z
                movecsr z,reg
}
#define NEWSIZE  0xffff
#define ZERO             0x0
main()
{
        setmemreg(CSR_LMSIZE,ZERO,NEWSIZE);
}
```

The assembly output generated from this example follows:

```
--      line 16, file ”e5.c”
        .beginai _setmemreg
        .airegrw        sr0     -- %tmpreg z
        -- .aireg?      lmsize  -- %csregnum reg
        .aibody
    ldlitlz 65535,sr0
    movecsr sr0,lmsize
        .endai          -- setmemreg
```

7.4. Examples for section 3.2.3.

The following example shows matching of several argument types.

```
asm void call(param)
{
        % const param;tmpreg x;
        -- insert(const)
                ldlitaddr param, x
        % tmpreg str;tmpreg x;
        -- insert(tmpreg)
                movereg param, x
}
int global_static_array[10];
int *global_static_ptr;main()
{
        int auto_array[10];
        int *auto_ptr;
        static int local_static_array[10];
        static int *local_static_ptr;

        call(auto_array);
        call(auto_ptr);
        call("string constant");
        call(global_static_array);
        call(global_static_ptr);
        call(local_static_array);
        call(local_static_ptr);
}
```

```
-- For main():
-- Prolog:
        push    fp
        alloc32 56,fp

.L49998:
        movereg fp,sr1
        add     -40,sr1
--      line 18, file "e6.c"
        .beginai _call
        .airegrw        sr0     -- %tmpreg x
        .airegrw        sr1     -- %tmpreg param
        .aibody
      -- insert(tmpreg)
     movereg sr1, sr0
        .endai          -- call
--      line 19, file "e6.c"
        .beginai _call
        .airegrw        sr0     -- %tmpreg x
        .airegrw        sr2     -- %tmpreg param
        .aibody
      -- insert(tmpreg)
     movereg sr2, sr0
        .endai          -- call
--      line 20, file "e6.c"
        .beginai _call
        .airegrw        sr0     -- %tmpreg x
        .ailab  .L59    -- %const param
```

```
          .aibody
        -- insert(const)
        ldlitaddr .L59, sr0

          .endai          -- call
--        line 22, file "e6.c"
          ldlit32l        _global_static_ptr,sr0
          ldlit32h        _global_static_ptr,sr0
--        line 21, file "e6.c"
          .beginai _call
          .airegrw        sr1     -- %tmpreg x
          .ailab  _global_static_array    -- %const param
          .aibody
        -- insert(const)
        ldlitaddr _global_static_array, sr1

          .endai          -- call
--        line 22, file "e6.c"
          ld.w    (sr0),sr1
          .beginai _call
          .airegrw        sr0     -- %tmpreg x
          .airegrw        sr1     -- %tmpreg param
          .aibody
        -- insert(tmpreg)
       movereg sr1, sr0
          .endai          -- call
--        line 24, file "e6.c"
          ldlit32l        .L58,sr0
          ldlit32h        .L58,sr0
--        line 23, file "e6.c"
          .beginai _call
          .airegrw        sr1     -- %tmpreg x
          .ailab  .L57    -- %const param
          .aibody
        -- insert(const)
        ldlitaddr .L57, sr1

          .endai          -- call
--        line 24, file "e6.c"
          ld.w    (sr0),sr1
          .beginai _call
          .airegrw        sr0     -- %tmpreg x
          .airegrw        sr1     -- %tmpreg param
          .aibody
        -- insert(tmpreg)
       movereg sr1, sr0
          .endai          -- call
.L49997:
.L56:
          release fp
          pop     fp
          ret
```

The following example shows part of the asm inline functions which are used in pathway control (courtesy of Harish Nag). The 'define' statements and asm inlines definitions come from the system include files provided for pathway control. The inlines show the use of the 'csregnum' storage class and illustrate how C defines should be used to make efficient use of the parameter passing mechanism. The defines allow the compiler to constant fold various masks so that *shift*s and *and*s are not done at the sight of the invocation (which might be in a loop).

```
/* asm.h */
/* typedefs */

typedef int pcti_type;
typedef int data_type;

/* cs registers */
#define stopsm    0x113
#define strm0r    0xC0
#define strm0w    0xC1
#define strm1r    0xC2
#define strm1w    0xC3
#define sysclki   0xAB
#define xbase     0x91

/* general defines */
#define VAL              0x01
#define RECV             0x00
#define STGATESELECT(gate,dir)                    \
                 (((dir)&0x1)|(((gate)&0x1)<<1))
#define STBINDSELECT(pcti,val)                    \
                 (((pcti)&0x1f)|(((val)&0x1)<<7))
#define bindrecv(pcti,gate)                               \
                 _bindrecv(STGATESELECT(gate,RECV),   \
                                 STBINDSELECT(pcti,VAL))

/* READCSREG */
asm data_type readcsreg(reg1) {
% csregnum reg1; tmpreg return;
        movecsr reg1, return
}
/* BINDRECV */
asm void _bindrecv(gateselect,bindselect) {
% register gateselect; register bindselect; void return;
        bind gateselect, bindselect
}
main()
{
        int x,y,z;
        /* BINDRECV*/
        printf("BINDRECV\n");
        (void)bindrecv(1,1);
        x=readcsreg(strm1r);
        printf("STRM1R=%d\n\n",x);
}
```

Parts of the assembly code that do not relate to the asm inline have been removed to save space.

```
_main:
--      line 48, file "h.c"
        ldlitlz  129,fv0  -- k=0x00000081(129)
        ldlitlz  2,fv1    -- k=0x00000002(2)
        bind fv1, fv0
--      line 49, file "h.c"
        movecsr strm1r, fv0
        st.w     fv0,+(fp,-4>>2)
```

You should only use the ‘const’ storage class if you have a special use for a constant value. If all you plan to do is “ldlit32”, then allow ‘tmpreg’ to match the constant. This has the following advantages:

1. The asm–insert will have fewer templates (easier to write and faster to compile).
2. The compiler will be able to load the constant into a register at any time before the asm–insert code (maybe even outside a loop).

Use ‘const’ when you have a template that will do something with the ‘const’ parameter other than immediately load it into a register.

For example:

```
asm int _reset_bit(bit,data) {
% const bit; tmpreg data; tmpreg return;
        bclr bit, data
% register bit; tmpreg data; tmpreg return;
        bclr bit, data
}
```

In this example, the writer knows that “bit” can be used directly in the “bclr” instruction but the compiler would not know that. Without the “const” template, the code produced would put the constant value into a temporary register and then do a “bclr”. If, on the other hand, the actual argument for “bit” was in a register, the register storage class would match (second leaf) and the incoming register would be used directly in the bclr instruction. This can be done because the “register” storage class guarantees to the compiler that the register is not written. In the case of the formal parameter “data”, the register *is* written so the storage class must be “tmpreg”. The compiler will insert a copy in case the value is used else where. But even in this case, the register allocator examines usage patterns and eliminates the move if possible.

In the next example, “_closeconmsgC” and “_closeconmsg” asm–inserts are shown. The _closeconmsgC insert handles all the matching possibilities explicitly. The second is the equivalent of the first except that all the ‘const’ and ‘tmpreg’ storage classes have been removed. The ‘const’ storage class is not needed because the expansion mechanism will automatically supply instructions to coerce constants into registers. The ‘tmpreg’ storage class is not needed because none of the parameters are needed for writing. The second (_closeconmsg) is preferred and should be used in place of the first.

```
asm void _closeconmsgC(data, gateselect) {
% register data; register gateselect;
        closeconmsgg data, gateselect
% register data; const gateselect; tmpreg temp1;
        ldlitaddr gateselect, temp1
        closeconmsgg temp1, gateselect
% register data; tmpreg gateselect;
        closeconmsgg data, gateselect
% const data; register gateselect; tmpreg temp1;
        ldlitaddr data, temp1
        closeconmsgg temp1, gateselect
% const data; const gateselect; tmpreg temp1; tmpreg temp2;
        ldlitaddr data, temp1
        ldlitaddr gateselect, temp2
        closeconmsgg temp1, temp2
% const data; tmpreg gateselect; tmpreg temp1;
        ldlitaddr data, temp1
        closeconmsgg temp1, gateselect
% tmpreg data; register gateselect;
        closeconmsgg data, gateselect
% tmpreg data; const gateselect; tmpreg temp1;
        ldlitaddr gateselect, temp1
        closeconmsgg data, temp1
% tmpreg data; tmpreg gateselect;
        closeconmsgg data, gateselect
}

asm void _closeconmsg(data, gateselect) {
% register data; register gateselect;
        closeconmsgg data, gateselect
}

int s1, s2, s3;

main()
{
  register int r1, r2, r3;
  int a1, a2, a3;

  _closeconmsgC( 992211, 33 );
  _closeconmsg( 992211, 33 );

  _closeconmsgC( 883322, a2 );
  _closeconmsg( 883322, a2 );

  _closeconmsgC( 294433, r2 );
  _closeconmsg( 294433, r2 );

  _closeconmsgC( 895544, s2 );
  _closeconmsg( 895544, s2 );

  _closeconmsgC( a1, 78244 );
  _closeconmsg( a1, 78244 );

  _closeconmsgC( a1, a2 );
  _closeconmsg( a1, a2 );
```

```
    _closeconmsgC( a1, r2 );
    _closeconmsg( a1, r2 );

    _closeconmsgC( a1, s2 );
    _closeconmsg( a1, s2 );

    _closeconmsgC( r1, 389655 );
    _closeconmsg( r1, 389655 );

    _closeconmsgC( r1, a2 );
    _closeconmsg( r1, a2 );

    _closeconmsgC( r1, r2 );
    _closeconmsg( r1, r2 );

    _closeconmsgC( r1, s2 );
    _closeconmsg( r1, s2 );

    _closeconmsgC( s1, 292866 );
    _closeconmsg( s1, 292866 );

    _closeconmsgC( s1, a2 );
    _closeconmsg( s1, a2 );

    _closeconmsgC( s1, r2 );
    _closeconmsg( s1, r2 );

    _closeconmsgC( s1, s2 );
    _closeconmsg( s1, s2 );

}
```

7.5. Examples for section 3.2.4.

The following example shows the use of the ‘label’ control line directive. The labels must be defined using the ‘lab’ control line directive.

```
asm max(x,y)
{
        % tmpreg return; register x,y;lab other,done;
                cmp.w x,y
                brif ilu.pos,other
                movereg x,return
                br done
        other:  movereg y,return
        done:
}
main()
{
        register int a=2,b=1,c;

        c = max(a,b);
        printf("max of %d and %d is %d\n",a,b,c);
}
```

The following shows assembly code generated for this example with unrelated sections deleted to save space. The inline mechanism generates labels that are unique for the assembly module. In this case, the labels are ".LASM1" and ".LASM2".

```
--      line 16, file "a4.c"
        ldlit32 2, sr1
        ldlit32 1, sr0
        ldlit32 2, ar1
        ldlit32 1, ar2
        ldlit32l        .L56,ar0
        .beginai _max
        .airegrw        ar3     -- RETURN: %tmpreg return
        .ailab  .LASM1  -- %lab
        .ailab  .LASM2  -- %lab
        .airegr sr0     -- %register y
        .airegr sr1     -- %register x
        .aibody
                cmp.w sr1,sr0
                brif ilu.pos,.LASM1
                movereg sr1,ar3
                br .LASM2
        .LASM1: movereg sr0,ar3
        .LASM2:
        .endai          -- max
        ldlit32h        .L56,ar0
        clr     ap
        call    _printf
```

The most common use for the **% use** directive is to designate either the special communication registers or the memory transfer registers used in the C&A instruction. The following example shows an asm–insert that is used to store values into local memory from the pathway.

```
asm void asm_receive_block0(num, data) {
% register num, data; tmpreg ptr; use ga0;
        movereg data, ptr
        loop num
        .beginloop .LOOP
        el st ga0,(ptr,4)+=
        .endloop .LOOP
}
```

The following example shows a case where the programmer handles some special integer cases himself and otherwise calls an external function to handle the general case. This is a viable strategy when there are a few cases that can be explicitly handled.

```
extern int real_wierd_int_func();
asm wierd_int_func(x)
{
% const(0) x; tmpreg return;
         ldlitlz 1,return
% const(1) x; tmpreg return;
         ldlitlz 0,return
% call real_wierd_int_func;
}
main()
{
         int y;
         y = wierd_int_func(0);
         y = wierd_int_func(4);
}
--      line 13, file ”a9.c”
        .beginai _wierd_int_func
        .aicall         -- %call not used, but present
        .airegrw        pr0     -- RETURN: %tmpreg return
        .aibody
        ldlitlz 1,pr0
        .endai          -- wierd_int_func
--      line 14, file ”a9.c”
        ldlit32 4, ar0
        clr     ap
        call    _real_wierd_int_func
```

7.6. Examples for section 4.

7.7. Longer examples

The following substantial example was supplied by HCR (the output is not shown because of its length).

```
/*
 * PROPRIETARY INFORMATION. Not to be reproduced,
 * transmitted, or disclosed in any way without permission.
 *
 * Produced by HCR Corporation,
 * 130 Bloor Street West, Toronto, Ontario, Canada.
 *
 * MODULE:       @(#)ptex3.c      2.3
 *
 * DATE: 89/03/2213:21:31
 *
 * AUTHORS: iWarp C:     Lorne Wilkinson
 *
 * REVISIONS:
 *
 */

#ifdef  SCCS    /*      Include SCCS id strings. */
static char SCCS_ID[] = "@(#)ptex3.c      2.3 HCR Corporation
89/03/22 13:21:31";
#endif  /* SCCS  */

#ifdef  IDENT   /*      #ident supported.        */
#ident  "@(#)ptex3.c     2.3     89/03/22 13:21:31"
#endif  /* IDENT */

/* Positive test: example.
 * Based on one of the exmples from Intel,
 * Consider this a realistic use of the insert mechanism.
 */

#define MASK    0x0f7e0000
#define ERRBIT  0x00800000

asm fp_event(ev)
/* Set the FPAEVT control space register, returning
 * the previous value of that register.
 * The value of the argument is checked to ensure it
 * is in range, if not the FPAEVT register is not
 * changed, and -1 is returned.
 */
{
%       void return; const(0) ev; tmpreg scratch;
        -- leaf 1
        ldlitlz 0,scratch
        movecsr scratch,fpaevt

%       tmpreg return; const(0) ev; tmpreg scratch;
        -- leaf 2
        ldlitlz 0,scratch
        movecsr fpaevt,return
        movecsr scratch,fpaevt
```

```
%       void return; const(0x00020000:0x0ffe0000) ev; lab \
erange; tmpreg scratch1, scratch2;
        -- leaf 3
        ldlithz (ev>>16),scratch1
        ldlithz (MASK>>16),scratch2
        and.w   scratch2,scratch1
        cmp.w   0,scratch1
        brif    ilu.zero,erange
        movecsr scratch1,fpaevt
erange:

%       tmpreg return; const(0x00020000:0x0ffe0000) ev;lab \
erange, out; tmpreg scratch1, scratch2;
        -- leaf 4
        ldlithz (ev>>16),scratch1
        ldlithz (MASK>>16),scratch2
        and.w   scratch2,scratch1
        cmp.w   0,scratch1
        brif    ilu.zero,erange
        movecsr fpaevt,return
        movecsr scratch1,fpaevt
        br out
erange:
        set     return
out:

%       void return; const ev; error;
        -- constant for event out of range, generate a
        -- compile time error

%       tmpreg return; const ev; error;
        -- constant for event out of range, generate a
        -- compile time error

%       void return; register ev; lab erange, setfpaevt;  \
tmpreg scratch1, scratch2;
        -- leaf 5
        cmp.w   0,ev
        brif    ilu.zero,setfpaevt
        movereg.w       ev,scratch1
        ldlithz (MASK>>16),scratch2
        and.w   scratch2,scratch1
        cmp.w   0,scratch1
        brif    ilu.zero,erange
setfpaevt:
        movecsr scratch1,fpaevt
erange:
```

```
%       tmpreg return; register ev; lab erange, out,  \
setfpaevt; tmpreg scratch1, scratch2;
        -- leaf 6
        cmp.w   0,ev
        brif    ilu.zero,setfpaevt
        movereg.w       ev,scratch1
        ldlithz (MASK>>16),scratch2
        and.w   scratch2,scratch1
        cmp.w   0,scratch1
        brif    ilu.zero,erange
setfpaevt:
        movecsr fpaevt,return
        movecsr scratch1,fpaevt
        br out
erange:
        set     return
out:
%       void return; tmpreg ev; lab erange, setfpaevt;    \
tmpreg scratch1;
        -- leaf 7
        cmp.w   0,ev
        brif    ilu.zero,setfpaevt
        ldlithz (MASK>>16),scratch1
        and.w   scratch1,ev
        cmp.w   0,ev
        brif    ilu.zero,erange
setfpaevt:
        movecsr ev,fpaevt
erange:
%       tmpreg return; tmpreg ev; lab erange, out,     \
setfpaevt; tmpreg scratch1;
        -- leaf 8
        cmp.w   0,ev
        brif    ilu.zero,setfpaevt
        ldlithz (MASK>>16),scratch1
        and.w   scratch1,ev
        cmp.w   0,ev
        brif    ilu.zero,erange
setfpaevt:
        movecsr fpaevt,return
        movecsr ev,fpaevt
        br      out
erange:
        set     return
out:
%       error;
}
```

```
#define FPA_EV_CONERR   0x20000
#define FPA_EV_INERES   0x40000
#define FPA_EV_INVOP    0x400000

int     global = FPA_EV_INERES;

#include "error.h"

main()
{
        int     retval;
        register int    local = FPA_EV_INVOP;
        register int  errors = 0;

        fp_event(0);
        retval = fp_event(0);
        if (retval != 0)
                ++errors;

        fp_event(FPA_EV_CONERR);
        retval = fp_event(FPA_EV_CONERR);
        if (retval != FPA_EV_CONERR)
                ++errors;

        fp_event(local);
        retval = fp_event(local);
        if (retval != FPA_EV_INVOP)
                ++errors;

        fp_event(global);
        retval = fp_event(global);
        if (retval != FPA_EV_INERES)
                ++errors;

        fp_event(local&FPA_EV_INVOP);
        retval = fp_event(local&FPA_EV_INVOP);
        if (retval != FPA_EV_INVOP)
                ++errors;

        /* Negative tests. */

        global = local = ERRBIT;

        fp_event(0);
        retval = fp_event(ERRBIT);
        if (retval != -1)
                ++errors;

        fp_event(0);
        retval = fp_event(global);
        if (retval != -1)
                ++errors;

        fp_event(0);
        retval = fp_event(local);
        if (retval != -1)
                ++errors;
```

```
            erpt(errors, __FILE__);
            return(errors);
    }
```

8. Conclusion

The implementation of the assembly language inclusion mechanism as specified in this document provides the functionality to generate a fast and clean insertion of assembly into the stream of compile code of a C program. If inserts are developed that obey the C semantics for function calls and arguments, the impact on the effectiveness of optimization will be minimal. A language extension of this complexity and hazardous potential should only be utilized by those intimately familiar with the iWarp assembler, calling sequence, architecture, and outstanding issues discussed in this document.

A. Implementation Notes

Overview

The current implementation of the assembly language inclusion mechanism is derived from that found in the PCC2 386 cross–compiler. This implementation was superior to that of the generic PCC2 compiler. Due to the additional complexity required by the asm–inline defined in this document, major changes have been made to the mechanism which performs the lexical and syntactic analysis of the insert. All of this work is now performed by the front end of the compiler, the data required for matching and expansion of the insert is passed to the back end of the compiler via an intermediate file. The back end utilizes this information to perform matching, communication with the register allocation mechanism, and final expansion of the insert.

In order to provide the cleanest and fastest possible interface to the use of the insert assembly body the back end of the compiler treats the pseudo call as an operator with register resource needs that are determined by the matching semantics. The requirements are utilized by the normal register allocation mechanism for the expression containing the pseudo call as a whole.

Argument List Size

The total number of scratch registers required by all arguments and all locals for an insert must not exceed the number available for use during the evaluation of the expression that contains the pseudo call. A fatal error **MUST** occur, because the code generator will not be able to rewrite the pseudo call so that it can utilize local temporaries on the stack, which is what it would normally attempt to do.

Parameter Passing Inconsistencies[2]

The expansion of an insert with a 'register' storage class parameter will result in effectively a pass by reference, while the expansion of the same insert with a 'tmpreg' storage class parameter will be a pass by value, since the argument is placed into a scratch register. For example, given the insert:

```
asm foo(a)
{
        % register a;

        ldlitlz 0,a;
        % tmpreg a;
        ldlitlz 0,a;
}
```

Expansion of:

```
main()
{
        register int i;

        foo(i);
}
```

will result in 'i' actually changing. However, the expansion of:

2. The information in this and the following sections is ***very*** dependent optimization level and peculiarities of how the optimizer handles specific program structures.

```
main()
{
        int i;

        foo(i);
}
```

will not alter the value of ‘i’ since it’s value has been passed to the insert via a scratch register. This is contrary to C pass by value semantics, and should not be done.

Permanent Register Live/Dead Status: the effect on ‘use’

In PCC2, permanent registers are allocated by the front end and no information regarding live/dead permanent registers is maintained by the back end. As a result, use of the ‘use’ storage class on a control line may result in a redundant store and load if the permanent register is dead.

Register Usage in the Assembly Body

A permanent register is safe to use (read/write) in an assembly body, provided it’s use has been specified in a ‘use’ directive in the control line for the leaf. Otherwise, the user may corrupt a register allocated by the compiler. All iWarp user accessible hardware registers can be considered safe. The compiler does have knowledge of these registers, but they are only used in conjunction with the intrinsic functions. Use of any other register–file register in an assembly body should be considered unsafe, if not positively dangerous. *Do not modify these registers behind the compiler’s back.*

The ‘use’ Storage Class

Any iWarp register name may be used as an operand of this storage class, however, only permanent registers will actually be saved before the execution of the assembly body, and restored afterwards.

B. Grammar

```
asm_function_definition  :==
                          'asm' [ type_specifier ] asm_function_declarator asm_function_body

asm_type_specifier       :==
                          'void'
                          'char'
                          'short'
                          'int'
                          'long'
                          'unsigned'
                          'float'
                          'double'

asm_function_declarator  :==
                          asm_declarator '(' [ asm_param_list ] ')'

asm_param_list           :==
                          formal_param_id [',' formal_param_id ]*

asm_declarator           :==
                          identifier
                          '*' identifier

asm_function_body        :==
                          '{' asm_block '}'

asm_block                :==
                          [ asm_leaf ]*

asm_leaf                 :==
                          '%' [ control ] '\n' (expansion_line|blank_line)*

control                  :==
                          arg_def  [ '\' '\n' control ]
                          return_def
                          'error' ';'
                          'call' ';'
                          'call' identifier ';'
```

```
expansion_line          :==
                    text [ operand [ ',' operand ]* ]
                    label ':'
                    arbitrary_string

arg_def                 :==
                    [ class name [ ',' name ]* ';' ]*
                    'use' reg [ ',' reg ]* ';'
                    'lab' label [ ',' label ]* ';'

return_def              :==
                    'tmpreg' 'return' ';'
                    'dtmpreg' 'return' ';'

operand                 :==
                    formal_param_id
                    local_var_id
                    'return'

class                   :==
                    'register'
                    'dregister'
                    'tmpreg'
                    'dtmpreg'
                    'regnum'
                    'regnum(' integer ')'
                    'regnum(' integer ':' integer ')'
                    'regnum(' hexnum ')'
                    'regnum(' hexnum ':' hexnum ')'
                    'regnum(' reg ')'
                    'regnum(' reg ':' reg ')'
                    'const'
                    'const(' integer ')'
                    'const(' integer ':' integer ')'
                    'const(' hexnum ')'
                    'const(' hexnum ':' hexnum ')'
                    'csregnum'
                    'csregnum(' integer ')'
                    'csregnum(' integer ':' integer ')'
                    'csregnum(' hexnum ')'
                    'csregnum(' hexnum ':' hexnum ')'
```

name :==
formal_param_id
local_var_id

label :==
identifier

formal_param_id :==
identifier

local_var_id :==
identifier

reg :==
'sr0' .. 'sr15'
'pr0' .. 'pr55'
'ar0' .. 'ar15'
'fv0' .. 'fv3'
'fp'
'ap'
'ev0' .. 'ev7'
'sp0' .. 'sp15'
'lm0' .. 'lm5'
'ga0' .. 'ga3'
hexnum

hexnum :==
'0x'['0' .. '9'|'a' .. 'f'|'A' .. 'F']+

integer :==
[–]?['0' .. '9']+

blank_line :==
['\t' ' ']* '\n'