To: Distribution

From: Richard A. Barnes

Date: 08/17/79

Subject: FORTRAN REGISTER OPTIMIIZER

1. Introduction

This paper describes the general design of register optimization for the Multics Fortran compiler. The reader is assumed to be familiar with the terminology used in discussing optimization and to have some knowledge of the design of the present Multics Fortran loop optimizer. An obsolete document which contains early design notes for a PL/I loop optimizer and a glossary of optimization terminology can be found in >udd>m>rab>loop>opt.runout. A short glossary also appears at the end of this paper.

2. Goals

The goal of the Fortran register optimizer is to improve the efficiency of object code by loading frequently used loop invariant pointer and index values into registers before the beginning of the loop rather than on each iteration of the loop, and by keeping frequently used induction variables in index registers rather than in storage if they are only used for subscripting and loop testing. An induction variable is defined to be a scalar local integer variable that is updated once in a loop in one of the following ways:

- by incrementing it by a loop invariant value
- by assigning to it another induction variable

Besides the variables coded by the user, the register optimizer must consider induction variables created by the loop optimizer during strength reduction.

2.1 16K Problem

One problem with the code generated by present compilers is the inefficiency of access code for variables more than 16384 words away from the beginning of a storage region. Consider the following storage fragment:

```
common /foo/ dummy(16385),iarray(16384),jarray(16384)
do 10 i = 1,16384
iarray(i) = jarray(i)
10        continue
```

The code generated by today's compiler looks something like this:

```
          ldq       1,dl
          stq       i
loop:     ldq       i
          adq       16384,dl
          eax2      0,ql
          ldq       i
          adq       32768,dl
          epp7      pr418,*             foo
          ldq       pr710,ql            jarray
          stq       pr710,2             iarray
          aos       i
          ldq       i
          cmpq      16384,dl
          tmoz      loop
```

Note that each iteration executes 12 instructions. With the register optimizer I hope to get a sequence like this:

```
          epp7      pr418,*             foo
          adwp7     16384,du
          epp5      pr418,*             foo
          adwp5     32768,du
          lxl2      1,dl
loop:     ldq       pr510,2             jarray
          stq       pr710,2             iarray
          adlx2     1,du
          cmpx2     16384,du
          tmoz      loop
```

for a cost of 5 instructions per iteration. (A possible future improvement for this particular example of array assignment might be the replacement of the entire loop by an MLR instruction!)

2.2 Loop Control

Since most loops that appear in FORTRAN programs are do-loops, these loops are controlled by induction variables. If the test that ends a loop involves an induction variable that is being kept in an index register, it would be nice if the test were made using a cmpxn instruction. Unfortunately, this has

several problems. An index register is treated by the hardware as a signed 17-bit integer when used with compare instructions while we normally represent integers in storage as signed 35-bit integers. Yet we could easily generate index values greater than 131071, so it is not clear which transfer instruction to use after a compare. Also, the cmpxn instruction compares an index register with an upper halfword in storage while we normally store integers in fullwords.

In order to deal with this problem, if an induction variable is to be kept in an index register and used in a comparison, the optimizer must figure out its range. As a example, for the loop

```
        do 100 index = start, finish, by
        body
100     continue
```

where "by" is a positive constant and "start" and "finish" are integers, the following cases apply:

Case_1: -131072 <= index <= 131071, finish is a constant

```
        lxln    start
loop:   body
        adlxn   by,du
        cmpxn   finish,du
        tmoz    loop
```

Case_2: -131072 <= index <= 131071, finish is a variable

```
        ldq     finish
        qls     18
        stq     ftemp
        lxln    start
loop:   body
        adlxn   by,du
        cmpxn   ftemp
        tmoz    loop
```

Case_3: index >= 0, finish is a constant

```
        lxln    start
loop:   body
        adlxn   by,du
        cmpxn   finish,du
        tnc     loop
        tze     loop

        OR
```

```
        lxln    start
loop:   body
        adlxn   by,du
        cmpxn   finish+1,du
        tnc     loop
```

Case_4: index >= 0, finish is a variable

```
        ldq     finish
        qls     18
        stq     ftemp
        lxln    start
loop:   body
        adlxn   by,du
        cmpxn   ftemp
        tnc     loop
        tze     loop

        OR

        ldq     finish
        adq     1,dl
        qls     18
        stq     ftemp
        lxln    start
loop:   body
        adlxn   by,du
        cmpxn   ftemp
        tnc     loop
```

Case_5: index range unknown

For now, we'll handle this as Case 6. However, if we were to change the converter to either provide information about do statements or to produce a different quadruple set from that generated now for do loops, we could generate code similar to that of the GCOS Fortran Y compiler, where the trip count is kept in a seperate register and is adjusted so as to complete when the register has a zero value:

```
        lxlm    -max(ntrips,1)
        lxln    start
loop:   body
        adlxn   by,du
        adlxm   1,du
        tze     loop

        OR
```

```
          lcq       ntrips
          tmi       2,ic
          lcq       1,dl
          eaxm      0,ql
          lxln      start
loop:     body
          adlxn     by,du
          adlxm     1,du
          tze       loop
```

This can be done by a form of test replacement as long as the adjustment for a minimum of 1 test is made.

Case_6: index is used for something other than subscripting or loop testing

(We don't keep index in an index register.)

```
          ldq       start
          stq       index
loop:     body
          ldq       by,dl
          asq       index
          ldq       index
          cmpq      finish
          tmoz      loop
```

OR if index is not busy on exit from the loop, and the loop is a do-loop:

```
          ldq       start
loop:     stq       index
          body
          ldq       index
          adq       by,dl
          cmpq      finish
          tmoz      loop
```

(This latter case will probably not be tried in initial releases of the
register optimizer.)

3. Index_Value_Analysis

The code generator puts two types of nodes into its machine state model's index registers: temporary nodes and symbol nodes. A temporary node represents the value of an expression that is not just a reference to a variable or constant. Wherever a particular temporary node appears in the internal representation, it always represents the same value. The compiler distinguishes between temporaries that are to be used as input to other

operators and those that are to be used for indexing, converting the former to the latter by use of the SUBINDEX operator. Because of this, a temporary that represents an index value will not be an input to a comparison, and thus analysis of its range or use is unnecessary.

A symbol node represents a reference to a variable. Each time a particular symbol node appears in the internal representation, it may represent a different value. A symbol node may be used for subscripting if it represents a scalar integer variable. If we want to keep the value of a variable in an index register throughout a loop without storing it in the loop, one of two conditions must be satisfied:

- the variable's value must not change within the loop

  OR

- the variable must be an induction variable for the loop whose value is not needed upon loop exit and its only use must be for subscripting, its own incrementing, comparison against a loop invariant, and assignment to other variables that satisfy this condition. This condition ensures that the high-order 18 bits of the induction variable are not needed for any calculation.

For each variable that is supposed to satisfy the second condition and this is used in a comparison, the range of the variable must be calculated. If the variable's range does not satisfy one of cases 1 - 4 mentioned in Section 2.2, that variable is not considered to satisfy the condition.

How do we gather the information to make these determinations? For each loop we will have 2 bit strings: USED_AS_SUBSCRIPT and ALWAYS_USED_AS_SUBSCRIPT, with one bit per variable. The loops are scanned one at a time in an order such that the most deeply nested ones are scanned first. At the start of each loop's scan the USED_AS_SUBSCRIPT string is initialized to all zeroes and the ALWAYS_USED_AS_SUBSCRIPT string is initialized to ones for those scalar integer variables that are either invariant or induction variables and to zeroes for all other variables. If the loop contains any inner loops, the USED_AS_SUBSCRIPT strings of the inner loops are OR'd to the USED_AS_SUBSCRIPT string of this loop, and the ALWAYS_USED_AS_SUBSCRIPT strings of the inner loops are AND'd to the ALWAYS_USED_AS_SUBSCRIPT string of this loop. The operators in the flow units belonging to this loop but not to inner loops are scanned. For each operator the inputs, if they are symbols, are processed as follows:

- OPT_SUBSCRIPT, SUBSTR -- the bit corresponding to operand(3) (the variable offset) is turned on in USED_AS_SUBSCRIPT. If operand(1) is a string which might require an offset greater than 262143, turn off operand(3)'s bit in ALWAYS_USED_AS_SUBSCRIPT.

- incrementing operators -- if the increment is not a constant, turn off the bit corresponding to the incremented variable in ALWAYS_USED_AS__SUBSCRIPT.

- RELATIONAL OPERATOR OR JUMP_ARITHMETIC -- if neither operand is an induction variable, do nothing. Otherwise, for each induction variable, if the operand being compared against is not loop invariant, turn off the bit corresponding to the induction variable in ALWAYS_USED_AS_SUBSCRIPT; if the operand being compared against is loop invariant , then add this operator to the COMPARISON list for this loop.

- ASSIGN - if the target does not have its bit on in ALWAYS_USED_AS_SUBSCRIPT, turn off the bit of the source in ALWAYS_USED_AS_SUBSCRIPT; otherwise, add this operator to the ASSIGNMENT list for this loop.

- other operators -- turn off the bits of all inputs in ALWAYS_USED_AS_SUBSCRIPT.

Once all the operators have been scanned, the COMPARISON list must be processed. Processing the COMPARISON list means finding the range of each induction variable in the list. If the induction variable is neither known to be always nonnegative nor known to be between -131072 and +131071, the variable's bit in ALWAYS_USED_AS_SUBSCRIPT must be turned off.

Deducing the range of each induction variable in the COMPARISON list is one of the trickiest tasks of the register optimizer. For now I have chosen a scheme that errs on the side of safety -- that is, it doesn't always succeed in calculating a range, but it never calculates a range incorrectly. This scheme only works for valid FORTRAN programs, however. It requires that subscriptrange never occurs within a loop. Basically, the scheme is this: all subscripting uses of the induction variable in the flow unit of the comparison and in the flow units in the dominator chain back to and including the loop entry unit are examined. Each subscripted reference gives us a range, thusly:

```
lower_limit = - constant_offset [+ increment, if increment is negative]

  if array has constant extents
       then upper_limit = array_size - constant_offset - 1
                          [+ increment, if increment is positive]
       else upper_limit = 262143 - constant_offset
                          [+ increment, if increment is positive]
```

(Each subscripted reference is represented as an OPT_SUBSCRIPT operator with operand(1) as the array name, operand(2) the constant offset, and operand(3) as the variable offset, that is the induction variable. Some adjustment might have to be made when SUBSTR is implemented.)

If more than one subscripted reference of the induction variable is found in the backward scan, the resulting lower limit is the maximum of all lower limits calculated, and the resulting upper limit is the minimum of all upper limits found. This works because all subscripted references found must be evaluated before the comparison is reached on every iteration of the loop. Once the range has been calculated, it is remembered for use by the code generator and for use by the next phase of range analysis.

After the COMPARISON list has been processed, the ASSIGNMENT list must be processed. The object of this processing is to ensure that no variable with a bit on in ALWAYS_USED_AS_SUBSCRIPT is assigned to a variable with a bit off in that string. Thus, we won't have a problem with the missing 18 high-order bits while keeping an induction variable in an index register. A secondary consideration is that we want to pass on any range data calculated during COMPARISON list processing, so that all variables that assign to variables used in comparisons, have range data. To process the ASSIGNMENT list each assignment on the list must be scanned. If the source of the assignment is a variable whose bit in the ALWAYS_USED_AS_SUBSCRIPT string is off, that assignment is removed from the list. If the source is a variable whose ALWAYS_USED_AS_SUBSCRIPT bit is on, but the target is a variable whose ALWAYS_USED_AS_SUBSCRIPT bit is off, the source variable's bit is turned off, the fact that a change has occurred on this scan of the list is remembered, and the assignment is removed from the list. If both the source and target are variable whose ALWAYS_USED_AS_SUBSCRIPT bits are on and the target has range information calculated for it while the source does not, range information is propagated to the source variable, and the fact that a change has occurred on this scan of the list is remembered. Scanning of the list repeats until no change occurs during a scan.

Once ASSIGNMENT list processing completes, we can derive a list of candidates for index register assignment by intersecting the USED_AS_SUBSCRIPT and ALWAYS_USED_AS_SUBSCRIPT strings. These strings are made available for analyzing outer loops and for the register assignment phase.

4. Storage_Allocation

Storage allocation must be done before global register allocation because the locations assigned to variables may affect register usage. This is especially important in the case of variables that are located at an offset of more than 16384 words from the beginning of their storage regions. No changes need to be made to the current FORTRAN code generator's storage allocator. After storage allocation has been completed, sufficient information exists to analyze register usage.

5. Global_Register_Usage_Analysis_and_Allocation

The algorithms described in this section are run on a loop-by loop basis from the inside-out and from front-to-back. This ensures that the innermost loops get the most chance for improvement. Each loop is analyzed for register usage, and loop invariants and induction variables are allocated registers across the loop before the next loop is processed.

5.1 Global_Register_Usage_Analysis

Two classes of information must be obtained during usage analysis for a loop in order to facilitate global register allocation:

- the number of uses of any values that may be allocated a register globally
- the maximum number of registers that would normally be allocated to non-global values during normal linear code generation if register optimization did not occur

This information provides us with the number of registers available for global allocation and a rank ordering of the values that might be allocated a register.

In order to get this information, which involves finding out which operations erase registers as well as causing them to be loaded, it appears that the most reliable method is to simulate code generation with a different macro interpreter that runs off the quadruples and fort_opt_macros_ just like the real code generator's interpreter does. Only instead of emitting code, this new interpreter would be compiling statistics for use in global register allocation. This means that information on which pl1_operators_ entries erase which registers need be kept in only one place. A problem of using two such similar interpreters is in keeping them consistent. Perhaps it is possible for them to be merged or else to share some code by include files. This still needs to be worked out.

The usage analysis interpreter must handle index values and pointer values in different ways. As mentioned earlier, index values are represented either by temporary nodes or symbol nodes. Only loop invariant temporaries are eligible for global register allocation. A temporary is considered invariant in a loop in which it is used if the operator producing its value is evaluated outside that loop. Both loop invariant symbols and symbols representing elibible loop induction variables are eligible for global register allocation. A symbol is found to be loop invariant if its bit in loop.set is off. A variable is an eligible induction variable if its bit in both the loop's USED_AS_SUBSCRIPT and ALWAYS_USED_AS_SUBSCRIPT strings are on. It is clearly possible to associate a usage count for the loop with each eligible value.

Pointer values do not at present have a uniform representation. Since for most storage regions it is possible that up to 16 pointers might be needed (because storage regions are a maximum of 262144 words long, while the 15-bit address fields can address 16384 words in a nonnegative direction), one way to represent each storage region might be with a vector of pointer value nodes containing usage counts. Each common block's header node could point to such a vector, and there could also be a vector each for automatic and static storage. Vectors would not be needed for argument pointers or the argument list itself. In these cases there would be a single pointer to a pointer value node.

The register usage analysis interpreter would scan all the quadruples contained in flow units of the particular loop that are not contained in inner loops of that loop. If an inner loop is encountered during the scan, it is treated as a black box that preserves all registers assigned across the inner loop and erases the rest. Note that this interpreter assumes no registers have been globally allocated for the loop being scanned.

After the loop is scanned, the values eligible for global allocation are sorted in descending order by usage count within their two classes -- index values and pointer values.

5.2 Global_Register_Allocation

The allocation algorithm we will use is not optimal but does a pretty good job for most programs. It is an improvement of Busam and Englund's algorithm, which was used for FORTRAN Y. The algorithm is run separately for the pointer registers and the index registers.

Let REGS_AVAIL be the number of available registers, and let REGS_USED be the number of registers allocated for this loop, not counting inner loops. Initialize REGS_USED to max(max_regs_needed_locally,1). (The second argument of the max function must be different for index registers if EIS is used. This has not yet been designed.)

We now consider each candidate for global allocation in turn in descending order by usage count. The effect of allocating a register to a candidate in a loop in which the candidate has not been allocated a register would be to increment REGS_USED for that loop by one. Therefore, we determine the effect of such an allocation for this loop and all nested loops. If for any of the examined loops, REGS_USED would exceed REGS_AVAIL, the attempt to allocate a register to the candidate fails.

If the attempt does not fail, we actually perform the allocation. The candidate is allocated a register in all inner loops in which it does not already have a register, and REGS_USED for each of those loops is adjusted accordingly. The candidate is then allocated a register in the loop being processed and REGS_USED is incremented for this loop. A LOAD_PREG or LOAD_XREG quadruple to load the register is inserted in the loop's back target. This will be seen as a register use when the outer loop is later scanned. If the candidate is an induction variable which is compared to a non-constant, a quadruple to force the non-constant to an upper halfword is placed in the back target, and the comparison is changed to be against the output of the new quadruple.

This algorithm ensures that enough registers are left available for local assignment while attempting to maximize the number that may be globally assigned. Note, that in this phase a register is allocated to a value. When the code generator runs later, a specific register will be assigned to a value (and the value assigned to the register).

6. Code_Generator_Changes

The functional changes needed for the code generator are described fairly completely in this section. Not all the necessary data structure changes are described. Some familiarity with the present fort_optimizing_cg is assumed.

6.1 Data_Structures

The machine state node, loop node, and operand nodes all need some modifications.

6.1.1 The_Machine_State

The machine state node presently contains the following fields for each index or pointer register:

- type
  a code defining the type of item in the register, specifying that the register is empty, or specifying that a register is "reserved" (locked).

- variable
  for index registers this specifies the variable in the register. This has other uses in pointer registers.

- used
  the last location this register was used.

- offset
  for pointer registers this specifies the offset of the value of the pointer from the beginning of a storage area. Not used for index registers.

Two bits will be added to each register description:

- global
  "1"b means the register has been globally assigned. If the type field is greater than zero, then the register currently contains the value globally assigned to it.

- reserved
  "1"b means the register is reserved (locked). This register may not be selected for loading a new value. This function used to be in the type field.

The type field currently has the following meanings:

-1	RESERVED
0	EMPTY
+n	contains known value

The meaning of -1 will be changed to "contains unknown value".

The representation of the indicators in the machine state should also be changed. Presently there is an "indicators_valid" bit which indicates that the indicators correspond to the value in the eaq, and there is a set of indicator substates for the eaq. This should be changed so that there is a code saying which register, variable, or comparison between 2 values caused the indicators to be set, and the indicators should be independent of any eaq states.

6.1.2 The_Loop_Node

Loop nodes are created by the loop optimizer and contain or point to all information specific to a loop. They are accessible by going from the opt_statement node to the flow_unit node to the loop node.

The loop node will be changed to contain a machine state image. This image will indicate which registers have been globally assigned which values for the loop.

The loop node will also be the repository of range information for induction variables globally assigned to registers.

6.1.3 The_Operand_Nodes

The various operand nodes will have a "globally_assigned" bit to indicate that the operand has been globally assigned to an index register. This is used to speed up register searching.

6.2 Functional_Changes

The changes for various operations and processing are described here.

6.2.1 Label_Processing

A referenced label on an executable statement usually represents a merging of two or more flows of control and always marks the beginning of a flow unit. Because of this, additional processing is needed which code generation reaches a label to support register optimization.

If the flow unit headed by the label belongs to a different loop than the flow unit that was being compiled, all operands that were globally assigned to an index register in the previous loop must have their "globally_assigned" bits turned off. Otherwise, this step is bypassed.

The machine states corresponding to the various flows of control are then intersected as is presently done. If there is a backward reference to the statement label or it is referenced in an ASSIGN statement, all registers are marked as empty.

Finally, if any registers have been globally assigned for the loop that the labelled statement is in, the machine state is updated from the loop node's machine state image to indicate which registers have globally assigned values in them. At this time any operands that are in globally assigned index registers have their "globally_assigned" bits turned on.

The action taken in the previous paragraph assumes that all globally assigned values are preloaded before the loop starts, and that if a register is locked or erased (because of a subroutine call, for example), that the register will be refreshed with its globally assigned value before the completion of the flow unit containing the erasure.

6.2.2 Loading_a_Register_with_a_Value

As is done currently, if a register already contains the desired value, everything is fine and no action takes place except to update the "used" field in the machine state. The changes in function are in what happens if the value is not contained in a register.

If the value has been globally assigned to a register but is not currently in the register, that register is selected for reloading the value. If the register is in the "reserved" state, an error occurs.

If the value has not been globally assigned to a register, get_free_reg is called as usual to select a register for loading. get_free_reg must not select a register that has been globally assigned, nor must it select a "reserved" register.

6.2.3 Register_Reservation_and_Freeing

Registers are reserved either because the code generator (actually fort_opt_macros_) wants that register for a special preemptive purpose, or code is about to be emitted that invalidates the register contents, such as the code for a subroutine call. Registers are freed after the code generator has emitted the code for which it reserved the registers, usually by invoking a free_regs macro. Any register that was reserved is now marked "empty" except for permanently assigned registers.

Two procedures reserve registers: reserve_regs and base_man_load_pr. free_regs frees all reserved registers. reserve_regs implements the reserve_regs macroinstruction and reserves the specified registers for "unknown" purposes. base_man_load_pr implements the load_pr macroinstruction and reserves the specified pointer register after loading an address into it. It is planned that for FORTRAN 77 char mode reservation will be requested in order to make multiple operands simultaneously addressable for EIS instructions. The LOAD_PREG and LOAD_XREG operators proposed later for register optimization will also wish to reserve registers so that the loads will not be undone.

Because of these latest plans, the concept of a reserved register has been split off from the concept of an unknown value. reserve_regs and base_man_load_pr will both turn on the "reserved" bit and set the "type" field to -1 to indicate "unknown value". Reservation done for EIS addressing or LOAD_PREG and LOAD_XREG operators will turn on the "reserved" bit, but will leave the "type" field unchanged.

Free_regs will turn off the "reserved" bit for all reserved registers and will change the "type" fields that have a -1 value to 0 for "empty". Positive "type" fields will be left unchanged.

6.2.4 End_of_Flow_Unit_Processing_--_Refresh_regs

Unlike traditional operating systems, Multics does not have calling conventions that guarantee transparency; that is, subroutines on Multics do not save and restore any registers they modify. While this minimizes memory references, it causes problems for an optimizing compiler that don't exist on traditional operating systems. One the one hand, a compiler may wish to globally assign values to registers across loops, preloading the registers before the loops. On the other hand, calls to subroutines and pl1_operators_ may erase these frozen registers, thus counteracting the compiler's intentions.

There are three possible approaches that could be taken here. The first would be to say that any registers clobbered by pl1_operators_ or subroutine calls within a loop are unavailable for global register assignment. This greatly simplifies code generation, but it loses opportunities for optimization, especially if the more frequently executed paths in the loop would not have clobbered the registers. (Note that subroutine calls are considered to clobber all registers.) The second approach is to generate code so that the calling program saves (if necessary) and restores registers across each call. This could cause unnecessary and redundant register loading if there are many calls close together as in an I/O statement. The present FORTRAN compiler uses a strategy like that for pr1 and pr4, and its bad effects can be seen in today's object code for I/O statements.

We will take a third approach that is an adaptation of the second approach. As registers are erased by subroutine or pl1_operators_ calls, they will be restored only as needed as long as flow stays within the same flow unit. The mechanism for this "as needed" restoration has already been described in Section 6.2.2 (Loading a Register). When control reaches the end of a flow unit, any globally assigned registers whose values are not in the registers must be "refreshed" , that is these values must be restored to the registers. This ensures that the values are known to be in the globally assigned registers at the beginning of every flow unit of the loop. It will be unnecessary to save globally assigned registers before these calls due to

several mechanisms. First, all pointer values used in FORTRAN programs are constants, so they don't need to be saved. Secondly, all loop invariant index values are either symbols whose values were stored outside the loop when they were assigned or temporaries whose values were stored before entry to the loop due to the normal action that occurs when a label (compiler generated or otherwise) or forward jump is encountered. We need a third mechanism to minimize unnecessary storing for globally assigned induction variables. The plan here is to store the index register into the variable's storage whenever the variable's value changes within the loop. This storing of the value would only occur if the register were ever erased within the loop or any of its descendants. This information must be provided by the interpreter that does the global register usage analysis and should be kept in the loop node.

There will be a new subroutine, refresh_regs, whose job will be to refresh all globally assigned registers whose values are not in the register. It will be called at the beginning of label processing, since a label marks the join of two flow units, and it will be called when the refresh_regs macroinstruction is invoked in any of the macro procedures for the various jump operators, since the registers must be refreshed before the jumps. Because of this latter case, refresh_regs must be prepared to optionally preserve the indicators across a refreshing operation since loading index registers changes the indicators, since the jump could be conditional, and since the refreshing must occur after all expressions have been evaluated in the flow unit but before the actual transfer instructions are emitted.

Refresh_regs should call free_regs as the last thing it does so that no registers are left reserved at the end of a flow unit. This could occur because of the processing of LOAD_PREG or LOAD_XREG operators, which is described in the next section.

6.2.5 LOAD_PREG and LOAD_XREG

As mentioned earlier, the global register allocation phase inserts LOAD_PREG and LOAD_XREG operators in the back targets of loops for which values are to be globally assigned to registers. The assignment of a value to a specific register does not occur until the code generator actually encounters and processes the LOAD_PREG or LOAD_XREG quadruple. When one of these quadruples is encountered, the appropriare base_man or xr_man procedure is called to select and load a register with the globally assigned value. That register is then reserved until the end of the flow unit. (Refresh_regs will call free_regs.) Because of this reservation, LOAD_PREG and LOAD_XREG quadruples should only appear at the end of a flow unit.

After a register has been selected and loaded, the machine state images for the loop that is about to be entered and all descendant loops must be updated to show that the specific register has been globally assigned the specific value. This enables the machine state to be appropriately initialized with the globally assigned registers at the beginning of every flow unit of the loops.

6.2.6 Nonindexing Operations

There are 3 types of nonindexing operations that will be carried out in index registers for globally assigned values: incrementing, comparison against a loop invariant, and assignment to or from other induction variables. These operations will be briefly discussed.

6.2.6.1 Incrementing of Index Values

The processing for incrementing operations is straightforward. The only complications is that the new value must be stored if the register is ever erased in the loop or any of its descendants.

Code sequences follow:

STORAGE_ADD

```
adlxn     constant,du
[sxln     induction_var]
```

STORAGE_ADD_ONE

```
adlxn     1,du
[sxln     induction_var]
```

NEG_STORAGE_ADD

```
sblxn     constant,du
[sxln     induction_var]
```

6.2.6.2 Comparison of Index Values

Comparison must be against loop invariants. The major problem is whether to use transfer or tsx0 instructions that follow arithmetic comparisons or those that follow logical comparisons. Also, note that comparisons against constants use DU modification.

Case_1: -131072 <= index <= 131071

```
cmpxn     comparand              (in upper 18 bits)
arithmetic comparison indicators used
```

Case_2: index >= 0

```
cmpxn     comparand              (in upper 18 bits)
logical comparison indicators used
```

Hopefully, more refined methods can be found for index value analysis (Section 3) that will do a better job of computing index ranges than those I have presently proposed.

6.2.6.3 Assignment_to_or_from_Other_Induction_Variables

Since induction variables are updated either by incrementing them by a loop invariant or by assignment from another induction variable, index registers may be used for assignments of one induction variable to another. The cases break down as to whether source, target, or neither are kept in storage. The interesting case is when the source is in an index register and the target is in storage. If the resulting value might be used in a comparison, we want a fullword stored. How this is done depends on the range anlysis done earlier. If the value will not be used in comparison, only a halfword need be stored. Note, also, that storage is updated if the target is in a register, but the register is erased in the loop.

Case_1: Source in register, target in register

```
eaxm      0,n
[sxlm     induction_var]
```

Case_2: Source in storage, target in register

```
lxln      source
[sxln     induction_var]
```

Case_3: Source in register, target in storage

Case_3.1: No range data on target

```
sxln      target
```

Case_3.2: -131072 <= target <= 131071

```
eaq       0,n
qrs       18
stq       target
```

Case_3.3: target >= 0

```
eaq       0,n
qrl       18
stq       target
```

6.2.7 Indicator_Changes

These changes are not necessary for register optimization and can be delayed if time becomes an issue. Presently if a variable in the q is used in a comparison, the code generator thinks that the variable is no longer in the q after the comparison. This is due to the indicators being considered par of the [eal]q in the machine state model. While this model was satisfactory for fast compilation and no code optimization, it is not so good for an optimizing compiler.

Plans to change this have not been worked out in detail, but the idea would be to more approximate the actions of the hardware the way the PL/I compiler does. Hopefully, a more uniform method than that used by pl1 can be developed. Basically the machine state would remember whether the indicators reflect a value in a specific register, a value of a specific variable in storage, the result of comparing 2 nonzero values, or none of the above.

7. Problems

Beside the omission of a detailed plan for handling the indicators, there are several known problems that have not yet been resolved. The most important problem is efficient initialization code for globally assigned induction variables. The code produced by the present compiler, the algorithms discussed here, and the desired code for a given loop follow:

```
          do 100 i = 1,1000
          a(i) = b(i) + c(i)
100       continue
```

Case_1: Installed Compiler

```
          ldq       1,dl
          stq       i
loop:     lxl2      i
          fld       b,2
          fad       c,2
          fstr      a,2
          aos       i
          ldq       i
          cmpq      1000,dl
          tmoz      loop
```

Case_2: Algorithms Described Here

```
            ldq       1,dl
            stq       i
            lxl2      i
loop:       fld       b,2
            fad       c,2
            fstr      a,2
            adlx2     1,du
            cmpx2     1000,du
            tmoz      loop
```

Case_3: Desired Code

```
            lxl2      1,dl
loop:       fld       b,2
            fad       c,2
            fstr      a,2
            adlx2     1,du
            cmpx2     1000,du
            tmoz      loop
```

I think that some minor constant folding can be done during one of the passes over the quads to solve this problem.

Another problem is that I wish we could do a better job of index value analysis. The present algorithm depends on a program's not taking subscriptrange, and does not utilize any information about starting and finishing values. Of course, the need for index value analysis would disappear if we had fullword index registers or general purpose registers!

8. Glossary

back_target (of a loop): the flow unit nearest to a loop that must be entered before a loop is entered. Loop optimization often moves operations from a loop to its back target.

dominator (of a flow unit, F): the flow unit nearest to F through which control must pass before F is entered. Also called the back dominator or immediate back dominator.

flow_unit: one or more statements defining a sequence of code that has no intermediate branching points; it can only be entered at the beginning and left at the end

induction_variable (in a loop): a scalar nonaliased local integer variable that is only updated once in a loop, either by incrementing it by a loop invariant, or by assigning it the value of another induction variable. The control variable of a do-loop is an induction variable.