```
;=========================================================================
;
; BOOKKEEPER
;
; This module implements the bookkeeping part of trace scheduling -- that
; part which, given a trace and a schedule for the trace, unsplices the
; trace from the MI flow graph and splices in the schedule, generating
; compensation code and keeping track of live variables and USE/DEFs.
;
; (TR.INITIALIZE-BOOKKEEP)
;     Initializes this module by clearing out global state.
;
; (TR.BOOKKEEP)
;     Performs bookkeeping on the current trace and its schedule.   The
;     bookkeeping is broken up into several main passes.
;
; Each main pass is described in detail at its source; here is a summary of
; the passes:
;
; (BK.RECORD-THE-TRACE)
;     Records the trace on a vector for easy access later on.
;
; (BK.BUFFER-THE-TRACE)
;     Isolates the trace from the rest of the MI flow graph by splicing in
;     dummies on each edge entering and exiting the edge.    These dummies
;     act as sentinels and make the implementation much easier.
;
; (BK.COALESCE)
;     Coalesces the machine operations in each cycle of the schedule into
;     a compacted MI;  the compacted MIs are stored on a vector for easy
;     access.
;
; (BK.FIX-SUCCESSORS)
;     Sets the successors of each coalesced, compacted MI and generates
;     split compensation code and DEFs at each split.
;
; (BK.FIX-PREDECESSORS)
;     Sets the predecessors of each coalesced, compacted MI and generates
;     rejoin compensation code and USEs at each join.
;
; (BK.PROPAGATE-LIVE-VARIABLES)
;     Performs incremental live analysis on all the rejoin and split
;     compensation code and USE/DEFs generated during bookkeeping.
;
; (BK.MISCELLANEOUS-CLEAN-UP)
;     Miscellaneous clean ups not deserving of a separate pass.
;
;=========================================================================

(eval-when (compile load)
    (include trace:declarations) )

(declare (special
    *bk.use-def-mis*         ;*** List of all MIs representing DEFs and
                             ;*** USEs created for current schedule.
    *bk.dummy-mis*           ;*** List of all dummy MIs created for the
                             ;*** current schedule.
    *bk.new-permanent-mis*   ;*** List of new non-dummy MIs made for current
                             ;*** schedule.
    *bk.rejoin-dummies*      ;*** List of dummy MIs bufferring rejoins.
    *bk.split-dummies*       ;*** List of dummy MIs buffering splits.
    *bk.cycle:mi*            ;*** A vector of coalesced MIs corresponding
                             ;*** to *TR.SCHEDULE*.
```



```
    *bk.trace-pos:mi*        ;*** A vector of elements on the trace.
    *bk.trace-pos:rejoin-cycle*
                             ;*** A vector for deciding where rejoins go.
    *bk.use-def-count*       ;*** Count of the USEs and DEFs spliced into
                             ;*** the MI graph.
    ) )


(defun tr.initialize-bookkeep ()
    (:= *tr.rejoin-count*         0)
    (:= *tr.split-count*          0)
    (:= *tr.partial-rejoin-count* 0)
    (:= *tr.partial-split-count*  0)
    (:= *bk.use-def-count*        0)

    (:= *bk.dummy-mis*         () )
    (:= *bk.rejoin-dummies*    () )
    (:= *bk.split-dummies*     () )
    (:= *bk.new-permanent-mis* () )
    (:= *bk.use-def-mis*       () )

    (:= *bk.cycle:mi*               () )
    (:= *bk.trace-pos:mi*           () )
    (:= *bk.trace-pos:rejoin-cycle* () )
    () )



(defun tr.bookkeep ()
    (tr.initialize-bookkeep)
    (bk.record-the-trace)
    (bk.buffer-the-trace)
    (bk.coalesce)
    (bk.fix-successors)
    (bk.fix-predecessors)
    (bk.propagate-live-variables)
    (bk.miscellaneous-clean-up)
    () )



;***======================================================================
;***
;*** (BK.RECORD-THE-TRACE)
;***
;*** Records the trace in *TR.TRACE-MIS* in the array *BK.TRACE-POS:MI*.
;*** The :TRACE-PRED and :TRACE-SUCC fields of each trace MI are set to
;*** be the predecessor and successor trace MIs.  The :TRACE-PRED of the
;*** first MI is set to be () (causing all of its predecessors to be
;*** treated as off-trace predecessors).  The :TRACE-SUCC of the last
;*** MI is set to be its first successor (just so that we have both an
;*** on-trace and off-trace successor for later code).
;***
;***======================================================================

(defun bk.record-the-trace ()
    (:= *bk.trace-pos:mi* (makevector (+ 1 *tr.trace-size*) ) )

    (loop (for trace-mi in *tr.trace-mis*)
          (incr trace-pos from 1)
          (initial prev-trace-mi () )
    (do
        (:= ([] *bk.trace-pos:mi* trace-pos) trace-mi)
```

```
        (:= (mi:trace-pos   trace-mi) trace-pos)
        (:= (mi:trace-pred  trace-mi) prev-trace-mi)
        (:= (mi:trace-succ  trace-mi) () )
        (if prev-trace-mi (then
            (:= (mi:trace-succ prev-trace-mi) trace-mi) ) ) )
    (next prev-trace-mi trace-mi) )

    (let ( (last-mi ([] *bk.trace-pos:mi* *tr.trace-size*) ) )
        (:= (mi:trace-succ last-mi) (car (mi:succs last-mi) ) ) )

    () )


;***=========================================================================
;***
;*** (BK.BUFFER-THE-TRACE)
;***
;*** This "buffers" the trace with dummy MIs that act as algorithmic
;*** sentinels.  A dummy is spliced in between every conditional jump
;*** on the trace and its off-trace successor.  A dummy is also spliced
;*** in between a join point in the trace and all the MIs that jump to
;*** that single point.  The edge into the beginning of the trace is
;*** considered a rejoin and buffered as such; all the edges leaving the
;*** last MI of the trace are considered off-trace split edges and also
;*** buffered.
;***
;*** One of the dummies joining to the beginning of the trace is
;*** distinguished as the "on-trace" predecessor of the first trace MI.
;*** Likewise, one of the dummies buffering the edges from the end is
;*** distinguished as the "on-trace" successor of the last MI.  This
;*** guarrantees that every conditional jump on the trace has an on-trace
;*** and off-trace successor.
;***
;*** But isolating the trace with dummies, we are guarranteed that every
;*** flow edge coming into and out of the trace has its tail or head off
;*** the trace, and that every trace MI is jumped to by at most one
;*** off-trace predecessor.
;***
;*** This buffering makes several things tractable, e.g. the handling
;*** of jumps on the trace that rejoin back to the trace and the merging
;*** of N+1-way jumps.
;***
;*** Rejoin dummies have the outgoing live variables stored in the
;*** :COPY-LIVE-OUT field.  Similarly, split dummies have the incoming
;*** live variables stored in :COPY-LIVE-OUT.  (Do we need the rejoin
;*** live info?)  After producing the split and rejoin copies, we propagate
;*** the live info from the split dummies up into the split copies.
;***
;***=========================================================================

(defun bk.buffer-the-trace ()

    (loop (incr trace-pos from 1 to *tr.trace-size*)
          (bind trace-mi ([] *bk.trace-pos:mi* trace-pos) )
    (do
        (if (|| (mi:rejoin? trace-mi)
                (== 1 trace-pos) )
        (then
            (bk.mi:buffer-rejoin trace-mi) ) )

        (if (mi:cond-jump? trace-mi) (then
            (bk.mi:buffer-split trace-mi (mi:off-trace-direction trace-mi))))))
```

```
    (let ( (first-mi  ([] *bk.trace-pos:mi* 1) )
           (last-mi   ([] *bk.trace-pos:mi* *tr.trace-size*) ) )

        (bk.mi:buffer-split last-mi 'left)

        (:= (mi:trace-pred first-mi)
            (car (mi:preds first-mi) ) )
        (:= (mi:trace-succ last-mi)
            (car (mi:succs last-mi) ) ) )
    () )


;***=========================================================================
;***
;*** (BK.MI:BUFFER-REJOIN REJOINED-MI)
;***
;*** Inserts a buffering dummy between an MI on the trace, REJOINED-MI,
;*** and all of its off-trace predecessors.  The live information is
;*** recorded on the dummy.
;***
;***=========================================================================

(defun bk.mi:buffer-rejoin ( rejoined-mi )
    (let ( (dummy (bk.mi:new-dummy) ) )
        (push *bk.rejoin-dummies* dummy)
        (bk.splice (mi:off-trace-preds rejoined-mi)
                   dummy
                   rejoined-mi)
        (:= (mi:copy-live-out dummy) (mi:live-in rejoined-mi) )
        () ) )


;***=========================================================================
;***
;*** (BK.MI:BUFFER-SPLIT SPLIT-MI DIRECTION)
;***
;*** Inserts a buffering dummy between a condition jump MI on the trace,
;*** SPLIT-MI, and one of its successors;    DIRECTION (LEFT or RIGHT)
;*** specifies which successor.    The live information is recorded on
;*** the dummy.
;***
;***=========================================================================

(defun bk.mi:buffer-split ( split-mi direction )
    (let ( (dummy     (bk.mi:new-dummy) )
           (edge-live (mi:live-out-on-edge split-mi direction) ) )
        (push *bk.split-dummies* dummy)
        (bk.splice (list split-mi)
                   dummy
                   (if (== 'left direction)
                       (car  (mi:succs split-mi) )
                       (cadr (mi:succs split-mi) ) ) )
        (:= (mi:copy-live-out dummy) edge-live)
        () ) )

;***=========================================================================
;***
;*** (BK.COALESCE)
;***
;*** This takes the schedule of machine operations and for each cycle
;*** in the schedule constructs a compacted MI that represents the machine
;*** operations of the cycle.  Each such "coalesced" MI is marked as
;*** compacted and placed in the vector mapping *BK.CYCLE:MI* (the mapping
```

```
;*** is 1-based, of course).
;***
;*** An empty, permanent compacted MI is always added to the end of the
;*** schedule.  This gives rejoins to the "end of the schedule" a place
;*** to jump to and insures that even empty schedules are reprsented by
;*** one compacted MI (helping to maintain the assertion that all the
;*** predecessors of DEFs and all the successors of USEs are compacted).
;***
;***=========================================================================

(defun bk.coalesce ()
    (:= *bk.cycle:mi* (makevector (+ 2 *tr.schedule-size*) ) )

        ;*** Coalesce each cycle of the schedule.
        ;
    (loop (incr cycle from 1 to *tr.schedule-size*) (do
        (:= ([] *bk.cycle:mi* cycle)
            (bk.mi:coalesce (schedule:[] *tr.schedule* cycle) ) )

        (loop (for mi in (mi:constituents ([] *bk.cycle:mi* cycle) ) ) (do
            (if mi (then
                (if (! (mi:first-cycle mi) )
                    (:= (mi:first-cycle mi) cycle) )
                (:= (mi:last-cycle mi) cycle) ) ) ) ) ) )

        ;*** Add the an empty permanent MI to the end of the schedule.
        ;*** This MI is made to point at the on-trace dummy buffering
        ;*** the end of the trace, and the dummy made to point at the
        ;*** MI.  This is done here since it won't follow out nicely in
        ;*** later passes.
        ;
    (let ( (last-mi    (bk.mi:coalesce () ) )
           (last-dummy (mi:trace-succ ([] *bk.trace-pos:mi* *tr.trace-size*))))
        (:= ([] *bk.cycle:mi* (+ 1 *tr.schedule-size*) )
            last-mi)
        (:= (mi:succs last-mi)    (list last-dummy) )
        (:= (mi:preds last-dummy) (list last-mi) ) )

    () )


;***=========================================================================
;***
;*** (BK.FIX-SUCCESSORS)
;***
;*** Fixes all the successors of the coalesced MIs on the schedule,
;*** splicing in split compensation code and DEFs where necessary.
;***
;***=========================================================================

(defun bk.fix-successors ()

    (loop (incr cycle from 1 to *tr.schedule-size*) (do
        (bk.schedule-mi:fix-succs ([] *bk.cycle:mi* cycle) cycle) ) )

    (let ( (last-mi ([] *bk.cycle:mi* (+ 1 *tr.schedule-size*) ) ) )
        (bk.splice-def&partial-schedule
            last-mi
            (schedule:split *tr.schedule* (+ 1 *tr.schedule-size*) )
            (car (mi:succs last-mi) ) ) )

    () )
```



```
;***=========================================================================
;***
;*** (BK.SCHEDULE-MI:FIX-SUCCS MI CYCLE)
;***
;*** Fixes the successors of the coalesced MI at a given cycle, generating
;*** split compensation code and a DEF if it is a split.
;***
;***=========================================================================

(defun bk.schedule-mi:fix-succs ( mi cycle )

(let ( (jump-mis (bk.mi:cond-jump-constituents mi cycle) ) )

        ;*** The successors of the new MI are those of the original
        ;*** conditional jumps, with the conditional jumps' trace
        ;*** successors replaced by the next scheduled instruction.  We
        ;*** update the predecessors of the off-trace MIs being jumped
        ;*** to, but not the predecessors of the on-trace MIs, which will
        ;*** be done in the next pass.
        ;
    (loop (for jump-mi in jump-mis)
          (bind succ-mi (mi:off-trace-succ jump-mi) )
          (initial succ-list () )
    (do
        (push succ-list succ-mi)
        (:= (mi:preds succ-mi) (top-level-substq mi jump-mi &&&) ) )
    (result
        (push succ-list ([] *bk.cycle:mi* (+ 1 cycle) ) )
        (:= (mi:succs mi) (dreverse succ-list) ) ) )

        ;*** Check to see if any elements were before the conditonal jump
        ;*** on the trace, but have now been scheduled below.  For each,
        ;*** call it ABOVE-MI, we make a copy of it for placement before
        ;*** each off-trace follower.  Note that these ABOVE-MIs are formed
        ;*** in reverse trace order, necessary since data precedence must
        ;*** be preserved in the sequence of copied instructions.  Thus
        ;*** the DECR in the loop below.  After the copies are made, we
        ;*** splice in the DEF and partial schedule of multi-cycle
        ;*** operations that spanned the split point that the code
        ;*** generator told us about.
        ;***
        ;*** Before splicing each split edge looks like:
        ;***
        ;*** mi -> dummy -> off-trace
        ;***
        ;*** After, it looks like:
        ;***
        ;*** mi -> partial-sched -> def -> split-copies -> dummy -> off-trace
        ;
    (loop (for  jump-mi      in jump-mis)
          (incr jump-number  from 1)
          (bind jumping-mi   mi
                jumped-to-mi (mi:off-trace-succ jump-mi)
                partial-schedule&def
                             (schedule:split *tr.schedule* cycle jump-number))
    (do
        (loop (decr cycle2 from (- (mi:trace-pos jump-mi) 1) to 1)
              (bind above-mi ([] *bk.trace-pos:mi* cycle2) )
        (do
            (if (&& (mi:first-cycle above-mi)               ;*** scheduled?
                    (> (mi:first-cycle above-mi) cycle) )
                (then
```

```
            (:= jumped-to-mi (bk.splice (list jumping-mi)
                                        (bk.mi:copy above-mi 'split)
                                        jumped-to-mi) )
            (++ *tr.split-count*)
            (bk.print-copy-after-message above-mi jump-mi) ) ) ) )

       (bk.splice-def&partial-schedule
           jumping-mi
           partial-schedule&def
           jumped-to-mi) ) )

   () ) )


;***=========================================================================
;***
;*** (BK.SPLICE-DEF&PARTIAL-SCHEDULE COALESCED-MI (DEF PARTIAL-SCHEDULE)
;***                                 OFF-TRACE-MI)
;***
;*** Splices a partial schedule and def as returned by SCHEDULE:SPLIT
;*** between a coalesced mi on the newly formed schedule and its off-trace
;*** successor.
;***
;***=========================================================================

(defun bk.splice-def&partial-schedule
       ( coalesced-mi (def partial-schedule) off-trace-mi )

   (let ( (jumped-to-mi off-trace-mi) )
       (if def (then
           (:= jumped-to-mi
               (bk.splice (list coalesced-mi)
                          (bk.mi:new-use-def def)
                          jumped-to-mi) ) ) )

       (if partial-schedule (then
           (:= *tr.partial-split-count*
               (+ &&& (schedule:length partial-schedule) ) )
           (loop (decr cycle from (schedule:length partial-schedule) to 1) (do
               (:= jumped-to-mi
                   (bk.splice (list coalesced-mi)
                              (bk.mi:coalesce (schedule:[] partial-schedule
                                                           cycle) )
                              jumped-to-mi) ) ) ) ) )

       jumped-to-mi) )



;***=========================================================================
;***
;*** (BK.PRINT-COPY-AFTER-MESSAGE COPIED-OP JUMP-OP)
;***
;*** Used for debugging.
;***
;***=========================================================================

(defun bk.print-copy-after-message ( copied-op jump-op )

   (if *tr.print-copying?* (then
       (msg 0 t t "A new copy of " (mi:source copied-op)
            ", schedule in cycles "
            (mi:first-cycle copied-op) ":" (mi:last-cycle copied-op) "." t
```



```
            "was produced to follow the jump "
            (mi:source jump-op)
            ", schedule in cycles "
            (mi:first-cycle jump-op) ":" (mi:last-cycle  jump-op) "."
            t) ) )
   () )


;***=========================================================================
;***
;*** (BK.FIX-PREDECESSORS)
;***
;*** Fixes all the predecessors of the coalesced MIs on the schedule,
;*** generating rejoin compensation code and USEs for MIs that are joins.
;***
;***=========================================================================

(defun bk.fix-predecessors ()

   (bk.build-trace-pos:rejoin-cycle)

       ;*** We can be sure that each MI is jumped to by at least its
       ;*** scheduled predecessor.  Recall that the last MI in the
       ;*** schedule is an empty MI.
       ;
   (loop (incr cycle from 2 to (+ *tr.schedule-size* 1) )
       (bind mi      ([] *bk.cycle:mi* cycle)
             pred-mi ([] *bk.cycle:mi* (- cycle 1) ) )
   (do
       (:= (mi:preds mi) (list pred-mi) ) ) )

       ;*** Now fix up all the normal rejoins, including the rejoin to the
       ;*** the top of the schedule.
       ;
   (loop (for rejoining-mi in *bk.rejoin-dummies*) (do
       (bk.rejoining-mi:fix-rejoin rejoining-mi) ) )

   () )



;***=========================================================================
;***
;*** (BK.REJOINING-MI:FIX-REJOIN REJOINING-MI)
;***
;*** Fixes a rejoin from the off-trace REJOINING-MI to somewhere in the
;*** schedule.  A rejoin point is selected at the highest point in the
;*** schedule such that all the operations scheduled below that point
;*** were also below the original rejoin point.  The rejoin from
;*** REJOINING-MI is made.  Then we copy into the rejoin all the operations
;*** that have moved in the schedule up above the rejoin point.  This
;*** is complicated when a conditional jump is copied -- we might have
;*** to copy some stuff onto the "off-trace" edge of the copied conditional
;*** jump as well.
;***
;*** Note how having a special empty MI at the end of the schedule makes
;*** things work out nice -- we always have a spot "at the end of the
;*** schedule" to make a rejoin to.
;***
;***=========================================================================

(defun bk.rejoining-mi:fix-rejoin ( rejoining-mi )

(let*( (trace-rejoined-mi (car (mi:succs rejoining-mi) ) )
```

```
      (rejoin-cycle      (bk.trace-mi:rejoin-cycle trace-rejoined-mi) )
      (rejoined-mi       ([] *bk.cycle:mi* rejoin-cycle) ) )

   (:= (mi:succs rejoining-mi) (list rejoined-mi) )
   (push (mi:preds rejoined-mi) rejoining-mi)

       ;*** Finally, figure out which operations were scheduled too early
       ;*** for the rejoin, but really need to be jumped to.  Make sure
       ;*** that they are copied before the rejoin, so they get executed
       ;*** too.  After the copies are made, we splice in the DEF and
       ;*** partial schedule of multi-cycle operations that spanned the
       ;*** join point that the code generator told us about.  Each join
       ;*** edge then looks like:
       ;***
       ;*** off-trace-->join-copies-->use-->partial-schedule-->rejoined-mi
       ;
   (loop (incr trace-pos from (mi:trace-pos trace-rejoined-mi)
                         to   *tr.trace-size*)
         (bind below-mi  ([] *bk.trace-pos:mi* trace-pos) )
   (do
       (if (&& (mi:first-cycle below-mi)   ;*** scheduled?
               (< (mi:last-cycle below-mi) rejoin-cycle) )
       (then
           (:= rejoining-mi (bk.copy-into-rejoin rejoining-mi
                                                 below-mi
                                                 trace-rejoined-mi
                                                 rejoined-mi
                                                 rejoin-cycle) ) ) ) ) )

   (bk.splice-use&partial-schedule
       rejoining-mi
       (schedule:join *tr.schedule* rejoin-cycle)
       rejoined-mi)
   () ) )

;***=========================================================================
;***
;*** (BK.COPY-INTO-REJOIN REJOINING-MI MI-TO-COPY TRACE-REJOINED-MI
;***                      REJOINED-MI REJOIN-CYCLE)
;***
;*** This copies a source MI up into a rejoin, doing the necessary extra
;*** copying for conditional jumps.
;***
;*** REJOINING-MI      - The MI jumping into the trace/schedule.
;*** MI-TO-COPY        - The MI to copy into the rejoin.
;*** TRACE-REJOINED-MI - The original MI in the trace that was joined to.
;*** REJOINED-MI       - The new MI in the schedule that is joined to.
;*** REJOIN-CYLE       - The cycle in the schedule of the rejoin.
;***
;*** The MI copied into the rejoin is returned.
;***
;***=========================================================================

(defun bk.copy-into-rejoin
       ( rejoining-mi mi-to-copy trace-rejoined-mi rejoined-mi rejoin-cycle )

(let ( (copied-mi (bk.mi:copy mi-to-copy 'join) ) )

    (bk.splice (list rejoining-mi) copied-mi rejoined-mi)

    (++ *tr.rejoin-count*)
    (bk.print-copy-before-message mi-to-copy trace-rejoined-mi)
```



```
        ;*** If a condition jump is copied up into the rejoin, then on
        ;*** the "off-trace" edge of the copy, we need to copy in all
        ;*** operations which were originally between the rejoin and the
        ;*** conditional on the trace, but haven't been copied up into
        ;*** the rejoin.
        ;
    (if (mi:cond-jump? mi-to-copy) (then
        (loop (incr trace-pos from (mi:trace-pos trace-rejoined-mi)
                              to   (- (mi:trace-pos mi-to-copy) 1) )
              (bind below-mi  ([] *bk.trace-pos:mi* trace-pos) )
              (initial pred-mi copied-mi)
              (when (&& (mi:first-cycle below-mi)    ;*** scheduled?
                        (! (< (mi:last-cycle below-mi) rejoin-cycle) ) ) )
        (do
            (:= pred-mi (bk.splice (list pred-mi)
                                   (bk.mi:copy below-mi 'join)
                                   (mi:off-trace-succ mi-to-copy) ) )
            (++ *tr.rejoin-count*)
            (bk.print-copy-before-message mi-to-copy trace-rejoined-mi))))

    copied-mi) )

;***=========================================================================
;***
;*** (BK.SPLICE-USE&PARTIAL-SCHEDULE USE PARITAL-SCHEDULE)
;***
;*** Splices a partial schedule and a use as returned by SCHEDULE:JOIN
;*** between an off-trace MI on a join and the coalesced MI that is joined
;*** to on the newly formed schedule.  Returns the MI in the join that
;*** now joins to the schedule (either a USE or else the last cycle of
;*** the partial schedule).
;***
;***=========================================================================

(defun bk.splice-use&partial-schedule
       ( off-trace-mi (use partial-schedule) coalesced-mi )

    (let ( (joining-mi off-trace-mi) )

        (if use (then
            (:= joining-mi
                (bk.splice (list joining-mi)
                           (bk.mi:new-use-def use)
                           coalesced-mi) ) ) )

        (if partial-schedule (then
            (:= *tr.partial-rejoin-count*
                (+ &&& (schedule:length partial-schedule) ) )
            (loop (incr cycle from 1 to (schedule:length partial-schedule) )
            (do
                (:= joining-mi
                    (bk.splice (list joining-mi)
                               (bk.mi:coalesce (schedule:[] partial-schedule
                                                            cycle) )
                               coalesced-mi) ) ) ) ) )

        joining-mi) )

;***=========================================================================
;***
```

```
;*** (BK.PRINT-COPY-BEFORE-MESSAGE COPIED-MI REJOIN-MI)
;***
;*** Used for debugging.
;***
;***=========================================================================

(defun bk.print-copy-before-message ( copied-mi rejoin-mi )
    (if *tr.print-copying?* (then
        (msg 0 t t "A new copy of "
              (mi:source copied-mi)
              ", schedule in cycles "
              (mi:first-cycle copied-mi) ":" (mi:last-cycle copied-mi)
              ", was produced to precede the rejoin at "
              (mi:source rejoin-mi) "."
              t) ) )
    () )


;***=========================================================================
;***
;*** (BK.TRACE-MI:REJOIN-CYCLE MI)
;***
;*** Finds the point in the schedule where we can make a rejoin that was
;*** originally to MI (an operation on the trace). That point is the
;*** highest point in the schedule below which are only operations which
;*** were originally below MI in the trace.
;***
;***=========================================================================

(defun bk.trace-mi:rejoin-cycle  ( mi )
    ([] *bk.trace-pos:rejoin-cycle* (mi:trace-pos mi) ) )


;***=========================================================================
;***
;*** (BUILD-TRACE-POS:REJOIN-CYCLE)
;***
;*** Builds the array *BK.TRACE-POS:REJOIN-CYCLE*, amazingly enough.
;*** T-P:R-C( t ) = c iff c is the earliest legal cycle in the schedule
;*** at which a join originally at trace position t can be made; e.g.
;*** given a join originally at trace position t, we make the join at
;*** cycle c in the schedule.  T-P:R-C is calculated by observing:
;***
;*** Let Tc be the minimum trace position of all the operations scheduled
;*** at cycle c or later.  Joins to trace positions after Tc must be remade
;*** at cycle c+1 or later.
;***
;***=========================================================================

(defun bk.build-trace-pos:rejoin-cycle ()
    (:= *bk.trace-pos:rejoin-cycle* (makevector (+ 1 *tr.trace-size*) ) )

    (loop (initial new-min-trace-pos *tr.trace-size*
                   min-trace-pos     *tr.trace-size*)
    (decr cycle from *tr.schedule-size* to 1)
    (do
        (loop (for elt in (mi:constituents ([] *bk.cycle:mi* cycle) ) )
              (when elt)
        (do
            (:= new-min-trace-pos
                (min new-min-trace-pos (mi:trace-pos elt) ) ) ) )

        (loop (incr trace-pos from (+ 1 new-min-trace-pos)
```



```
                              to   min-trace-pos)
        (do
            (:= ([] *bk.trace-pos:rejoin-cycle* trace-pos)
                (+ 1 cycle) ) ) )
        (:= min-trace-pos new-min-trace-pos) )
    (result
        (loop (incr trace-pos from 1 to min-trace-pos) (do
            (:= ([] *bk.trace-pos:rejoin-cycle* trace-pos) 1) ) ) ) )
    () )



;***=========================================================================
;***
;*** (BK.PROPAGATE-LIVE-VARIABLES)
;***
;*** This calculates live variables for all the split and join copies
;*** that were made.  The live info is propogated up from the buffering
;*** split dummies up into the split copies; and from the USEs at joins
;*** up into the join copies.
;***
;***=========================================================================

(defun bk.propagate-live-variables ()
    (let ( (to-do () ) )

            ;*** Set the :COPY-LIVE-OUT of all USE MIs to be the set of
            ;*** of names constained in the USE.
            ;
        (loop (for use-mi in *bk.use-def-mis*)
              (when (== 'use (mi:operator use-mi) ) )
        (do
            (:= (mi:copy-live-out use-mi)
                (for ( (var loc) in (oper:part (mi:oper use-mi) 'body) ) (save
                     var) ) ) ) )

            ;*** Initialize TO-DO to be all the predecessors of the split
            ;*** dummies and USE MIs that are uncompacted.  Those
            ;*** uncompacted predecessors are either copies or USE/DEFs.
            ;
        (loop (for split-mi in *bk.split-dummies*) (do
            (loop (for pred-mi in (mi:preds split-mi) )
                  (when (! (mi:compacted? pred-mi) ) )
                  (when (! (memq to-do pred-mi) ) )
            (do
                (push to-do pred-mi) ) ) ) )

        (loop (for use-mi in *bk.use-def-mis*)
              (when (== 'use (mi:operator use-mi) ) )
        (do
            (loop (for pred-mi in (mi:preds use-mi) )
                  (when (! (mi:compacted? pred-mi) ) )
                  (when (! (memq to-do pred-mi) ) )
            (do
                (push to-do pred-mi) ) ) ) )

            ;*** Repeatedly pick an MI from TO-DO, all of whose successors
            ;*** have known live info.  Caclulate the live info of that
            ;*** MI and remove it from TO-DO.  If the predecessor of MI
            ;*** is uncompacted, add it to TO-DO.  This loop could probably
            ;*** be made much more efficient if need be.
            ;
```

```
        (loop (while to-do)
              (initial mi () )
        (do
            (:= mi
                (loop (for possible-mi in to-do) (do
                    (if (for-every (succ-mi in (mi:succs possible-mi) )
                              (!== *tr.unknown-copy-live-out*
                                    (mi:copy-live-out succ-mi) ) )
                    (then
                        (return possible-mi) ) ) )
                (result () ) ) )

                ;*** We better have found an MI and it better be a copy
                ;*** or USE/DEF and it better have one predecessor.
                ;
            (assert (&& mi
                        (== 1 (length (mi:preds mi) ) )
                        (|| (mi:copy-type mi)
                            (memq (mi:operator mi) '(use def) ) ) ) )

            (:= (mi:copy-live-out mi)
                (loop (for succ-mi in (mi:succs mi) )
                      (reduce unionq () (mi:live-in succ-mi) ) ) )

            (:= to-do (top-level-removeq mi to-do) )
            (loop (for pred-mi in (mi:preds mi) )
                  (when (! (mi:compacted? pred-mi) ) )
                  (when (! (memq to-do pred-mi) ) )
            (do
                (push to-do pred-mi) ) ) ) )

        () ) )

;***=========================================================================
;***
;*** (BK.MISCELLANEOUS-CLEAN-UP)
;***
;*** Performs miscellaneous clean ups that aren't deserving of a separate
;*** pass.
;***
;***=========================================================================

(defun bk.miscellaneous-clean-up ()

        ;*** Dispose of the source MIs on the trace:
        ;
    (:= *tr.s* (set-diffq *tr.s* *tr.trace-mis*) )

        ;*** Dispose of the dummy list:
        ;
    (loop (for mi in *bk.dummy-mis*) (do
          (bk.unsplice mi) ) )
    (:= *tr.s* (set-diffq *tr.s* *bk.dummy-mis*) )
    (:= *bk.dummy-mis* () )

        ;*** Update the expect value of each new element:
        ;
    (loop (for mi in *bk.new-permanent-mis*) (do
          (mi:find-expect mi) ) )
    (:= *bk.new-permanent-mis* () )

        ;*** Remove any new USEs that have no uncompacted predecessors,
```



```
        ;*** and any DEFs that have no uncompacted successors.  They are
        ;*** useless.
        ;
    (loop (for use-def-mi in *bk.use-def-mis*) (do
        (if (for-every (mi in (if (== 'use (mi:operator use-def-mi) )
                                  (mi:preds use-def-mi)
                                  (mi:succs use-def-mi) ) )
                (mi:compacted? mi) )
        (then
            (-- *bk.use-def-count*)
            (bk.unsplice use-def-mi)
            (:= *tr.s* (top-level-removeq use-def-mi *tr.s*) ) ) ) ) )

    (:= *bk.use-def-mis* () )

        ;*** Clear out the constituents field, which contains only old
        ;*** MIs which we won't want to use again, so we can gc em.
        ;
    (loop (incr cycle from 1 to *tr.schedule-size*) (do
        (:= (mi:constituents ([] *bk.cycle:mi* cycle) ) () ) ) )

        ;*** Clear out the successors and predecessors of the trace elements
        ;*** just in case.
        ;
    (loop (for mi in *tr.trace-mis*) (do
        (:= (mi:preds mi) () )
        (:= (mi:succs mi) () )
        (:= (mi:trace-succ mi) () )
        (:= (mi:trace-pred mi) () ) ) )

        ;*** If requested, add (TRACE i) to each compacted instruction.
        ;
    (if *tr.generate-trace-info?* (then
        (loop (incr cycle from 1 to *tr.schedule-size*) (do
            (push (mi:source ([] *bk.cycle:mi* cycle) )
                  '(trace ,*tr.trace-number*) ) ) ) ) )

        ;*** Now we know how many new elements we've generated, so we
        ;*** update *tr.ops-left*.  Don't need the rejoin list anymore,
        ;*** either:
        ;
    (:= *tr.ops-left* (+ &&&
                         (+ *tr.rejoin-count*
                         (+ *tr.split-count*
                            *bk.use-def-count*) ) ) )
    (:= *tr.rejoin-total*         (+ &&& *tr.rejoin-count*) )
    (:= *tr.split-total*          (+ &&& *tr.split-count*) )
    (:= *tr.partial-rejoin-total* (+ &&& *tr.partial-rejoin-count*) )
    (:= *tr.partial-split-total*  (+ &&& *tr.partial-split-count*) )

    (:= *bk.rejoin-dummies* () )
    (:= *bk.split-dummies*  () )
    () )


;***=========================================================================
;***
;*** (BK.MI:COALESCE CYCLE-LIST)
;***
;*** Makes a new compacted MI representing the machine operations of a
;*** cycle in the schedule.  CYCLE-LIST represents a cycle in the schedule,
;*** and is is a list of pairs of the form:
;***
```

```
;***     (MACHINE-OPERATION SOURCE-MI)
;***
;*** where SOURCE-MI represents the source operation that caused
;*** MACHINE-OPERATION to be generated (SOURCE-MI may be () ).  The new
;*** MI is marked as compacted, its :SOURCE is set to be the list of
;*** machine operations in the cycle, its :CONSTITUENTS the SOURCE-MIs,
;*** and its :TRACE-DIRECTION to be a list of LEFT/RIGHT indicating which
;*** way the trace goes for that machine operation (significant only for
;*** jumps).
;***
;***=========================================================================

(defun bk.mi:coalesce ( cycle-list )
    (let ( (new-mi  (mi:new
                        number       (++ *tr.mi-number*)
                        trace        *tr.trace-number*
                        compacted?   t) ) )
        (push *tr.s*                 new-mi)
        (push *bk.new-permanent-mis* new-mi)

        (loop (for (oper source-mi) in cycle-list)
              (initial current-prob 1.0)
        (do
            (push (mi:source       new-mi) oper     )
            (push (mi:constituents new-mi) source-mi)

            (if (&& source-mi
                    (mi:cond-jump? source-mi) )
            (then
                (push (mi:trace-direction    new-mi)
                      (mi:on-trace-direction source-mi) )
                (:= current-prob
                    (* current-prob
                       (nr.edge-prob source-mi
                                     (mi:off-trace-succ source-mi) ) ) )
                (push (mi:edge-prob new-mi) current-prob) ) ) )

        (result
            (push (mi:edge-prob new-mi)
                  (- 1.0
                     (loop (for prob in (mi:edge-prob new-mi) )
                           (initial sum 0.0)
                           (next sum (+ sum prob) )
                        (result sum) ) ) )
            (:= (mi:edge-prob new-mi)
                (dreverse (mi:edge-prob new-mi) ) )
            (:= (mi:source new-mi)
                (dreverse (mi:source new-mi) ) )
            (:= (mi:constituents new-mi)
                (dreverse (mi:constituents new-mi) ) )
            (:= (mi:trace-direction new-mi)
                (dreverse (mi:trace-direction new-mi) ) ) ) )

        new-mi) )


;***=========================================================================
;***
;*** (BK.MI:COPY OLD-MI COPY-TYPE)
;***
;*** Makes a copy of an MI suitable for splicing somewhere else; none
;*** of the successor or predecessor fields are set.  The copy is recorded
;*** on *TR.S* and *bk.new-permanent-mis*.  COPY-TYPE should be either
```



```
;*** SPLIT or JOIN.
;***
;***=========================================================================

(defun bk.mi:copy ( old-mi copy-type )
    (let ( (new-mi  (mi:new
                        number       (++ *tr.mi-number*)
                        constituents (list old-mi)
                        source       (mi:source old-mi)
                        copy-type    copy-type
                        trace        *tr.trace-number*
                        compacted?   ()
                        edge-prob    (mi:edge-prob old-mi) ) ) )

        (push *tr.s*                 new-mi)
        (push *bk.new-permanent-mis* new-mi)
        new-mi) )



;***=========================================================================
;***
;*** (BK.MI:NEW-DUMMY)
;***
;*** Makes a new dummy MI which is marked as compacted.  The dummy is
;*** recorded on *TR.S* and *BK.DUMMY-MIS*.
;***
;***=========================================================================

(defun bk.mi:new-dummy ()
    (let ( (new-mi (mi:new
                        number       (++ *tr.mi-number*)
                        constituents ()
                        trace        *tr.trace-number*
                        compacted?   t) ) )

        (push *tr.s*          new-mi)
        (push *bk.dummy-mis* new-mi)
        new-mi) )


;***=========================================================================
;***
;*** (BK.MI:NEW-USE-DEF USE-OR-DEF)
;***
;*** Makes a new MI for a use or def.
;***
;***=========================================================================

(defun bk.mi:new-use-def ( use-or-def )
    (let ( (new-mi  (mi:new
                        number       (++ *tr.mi-number*)
                        source       (list use-or-def)
                        compacted?   ()
                        trace        *tr.trace-number*
                        edge-prob    '(1.0) ) ) )

        (:= *bk.use-def-count* (+ 1 *bk.use-def-count*) )
        (push *tr.s*                 new-mi)
        (push *bk.new-permanent-mis* new-mi)
        (push *bk.use-def-mis*       new-mi)
        new-mi) )
```

```
;***=========================================================================
;***
;*** (BK.MI:COND-JUMP-CONSTITUENTS MI MI-CYCLE)
;***
;*** Returns those source MI constituents of a a coalesced MI that are
;*** conditional jumps.  For these purposes, a multi-cycle cond-jump is
;*** considered a constituent of only its last cycle.
;***
;***=========================================================================

(defun bk.mi:cond-jump-constituents ( mi mi-cycle )
    (loop (for source-mi in (mi:constituents mi) )
          (initial result () )
    (do
        (if (&& source-mi
                (mi:cond-jump? source-mi)
                (== mi-cycle (mi:last-cycle source-mi) ) )
        (then
            (push result source-mi) ) ) )
    (result
        (dreverse result) ) ) )


;***=========================================================================
;***
;*** (BK.SPLICE PREDS TARGET SUCC)
;***
;*** Takes a list of args (PREDS), an MI (TARGET), and another
;*** MI, the SUCC.  It is assumed that each pred is in preds of succ.
;*** SPLICE replaces succ with the target on the succs of each pred.
;*** If succ is not a successor of some pred, then no change occurs for
;*** that pred.  The preds of succ is changed so that all of the
;*** elements of of pred are removed, and the target is inserted (I know,
;*** this is a dumb comment, just repeating the code...).
;***
;*** If TARGET is uncompacted, SPLICE assumes that we have a copy of a
;*** new MI which has just been built from a list of trace elements.  That
;*** is, it expects the elements of the list (MI:CONSTITUENTS TARGET)
;*** to have trace successors and preds.  If one of the elements of the
;*** list is a cond.  jump, the off trace flowsus are are preserved in
;*** the copy.
;***
;*** If TARGET is compacted, then we make SUCC be the successor of TARGET,
;*** always.
;***
;*** The effect of all this, and the corresponding changes to the
;*** preds&su of target, is to insert target in the flow between all
;*** the preds and the succ.  preds are a list while succ only one elt
;*** because it happened to be more convenient to use it that way.
;***
;***=========================================================================

(defun bk.splice (preds target succ)

    (assert (subset? preds (mi:preds succ) ) )
    (assert (not (memq target (mi:preds succ) ) ) )

        ;*** First have target replace succ as the successor of each
        ;*** element of pred.  Then make pred the predecessor list of
```



```
        ;*** target.
        ;
    (for (pred in preds) (do
        (:= (mi:succs pred)
            (top-level-substq target succ (mi:succs pred) ) ) ) )

    (:= (mi:preds target) preds)


        ;*** Remove all the preds from the predecessor list of succ, and
        ;*** replace them with target.
        ;
    (:= (mi:preds succ) (set-diffq (mi:preds succ) preds) )
    (push (mi:preds succ) target)


        ;*** Finally, make succ the successor of target.  If target is
        ;*** to represent an uncompacted conditional jump, it should still
        ;*** jump to all the off-trace elements, and we need to update
        ;*** their preds's.  We assume that we're only splicing 2-way
        ;*** jumps.
        ;
    (if (&& (! (mi:compacted? target) )
            (mi:constituents target) )
    (then
        (assert (<= (length (mi:constituents target) ) 1) )
        (let ( (jump-elt  (car (mi:constituents target) ) ) )
            (:= (mi:succs target)
                (top-level-substq succ
                                  (mi:trace-succ jump-elt)
                                  (mi:succs jump-elt) ) )
            (for (target-succ in
                    (top-level-removeq succ (mi:succs target) ) )
            (do
                (push (mi:preds target-succ) target) ) ) ) )
    (else
        (:= (mi:succs target)  (list succ) ) ) )

    target)


;***=========================================================================
;***
;*** (BK.UNSPLICE TARGET)
;***
;*** Undoes the effects of a splice of the element.  We don't need
;*** to give the preds and succ, since they can be determined from the target
;*** alone.  When the succs of the target is more than one element, there
;*** is an error; for some reason, I feel it necessary to test for this one,
;*** and report on it.
;***
;***=========================================================================

(defun bk.unsplice ( target )
    (assert (! (cdr (mi:succs target) ) ) )

    (let ( (preds (mi:preds target) )
           (succ  (car (mi:succs target) ) ) )

        (for (pred in preds) (do
            (:= (mi:succs pred)
                (top-level-substq succ target (mi:succs pred) ) ) ) )
```

```
(:= (mi:preds succ)
    (unionq preds (top-level-removeq target (mi:preds succ) ) ) )

(:= (mi:succs target) () )
(:= (mi:preds target) () )
() ) )
```

```
;** (build *tr.build-module-list*)
;** (build.compile *tr.build-module-list*)

(:= *tr.build-module-list* '(
    trace:mi

    trace:naddr-rec
    trace:mis-to-pnaddr

    trace:trace-picker
    trace:bookkeeper
    trace:display
    trace:compact
    trace:compact-options
    ) )

(:= *build-module-list* (append *build-module-list* *tr.build-module-list*) )
```

```
(eval-when (compile load)
    (include trace:declarations) )



;;;
;;; Initialization of all the trace modules.  This is all gross organization,
;;; but it will have to do until the bookkeeper is rewritten.
;;;

(defun tr.initialize ()
    (tr.initialize-bookkeep)            ;;; Initialize the bookkeeper.
    (tr.initialize-trace-picker)        ;;; Initialize the trace picker.
    (initialize-code-generator)         ;;; Initialize the code generator.

    (:= *tr.s*              () )       ;;; All known MIs.

    (:= *tr.mi-number*    0)           ;;; Counts the new mi's.
    (:= *tr.rejoin-total* 0)           ;;; How many rejoin MIs total
    (:= *tr.rejoin-count* 0)           ;;; How many rejoin MIs this trace
    (:= *tr.split-total*  0)           ;;; How many split MIs total
    (:= *tr.split-count*  0)           ;;; How many split MIs this trace
    (:= *tr.partial-rejoin-total* 0)   ;;; How many partial-schedule rejoin MIs
                                       ;;;     total
    (:= *tr.partial-rejoin-count* 0)   ;;; How many partial-schedule rejoin MIs
                                       ;;;     this trace
    (:= *tr.partial-split-total*  0)   ;;; How many partial-schedule split MIs
                                       ;;;     total
    (:= *tr.partial-split-count*  0)   ;;; How many partial-schedule split MIs
                                       ;;;     this trace
    (:= *tr.total-cycles*         0)   ;;; How many cycles in final sched
    (:= *tr.schedule* () )             ;;; Clear out any old schedule.
                                       ;;; Initialize the hook functions

    (if *tr.generate-code-hook*
        (funcall *tr.generate-code-hook* () ) )
    (if *tr.dag-hook*
        (funcall *tr.dag-hook* () ) )

    () )




(defun compact ( naddr )

(let ( (*tr.space-mode* *tr.space-mode*)   ;;; Dynamically bind these vars
       (*tr.window*     *tr.window*) )     ;;; in case we change them during
                                           ;;; compaction below -- dynamic
                                           ;;; binding will restore their values.

;;; Initialization (done once per setting of *tr.s*):

    (tr.initialize)
    (set-compactor-flags)             ;;; Sets flags for space-saving,
                                      ;;; display methodology, etc.

    (tr.get-*tr.s* naddr)                ;;; Builds the global list *tr.s*

    (:= *tr.start-length* (length *tr.s*));;; Saved for final print-out

    (:= *tr.ops-left*                    ;;; For running total of ops
        (loop (for elt in *tr.s*)        ;;; still needing compaction.
```



```
              (initial ops 0)
          (do
              (if (! (mi:compacted? elt) )
                  (++ ops) ) )
          (result ops) ) )

;;; Scheduling loop:

                                       ;;; supporting information.
    (loop (incr *tr.trace-number* from 1)
    (do
        (if (&& *tr.delayed-space-mode*
                (= *tr.trace-number* *tr.delayed-space-mode-trace*) )
        (then
            (msg 0 t t "Now using delayed space mode//window "
                 *tr.delayed-space-mode* "//" *tr.delayed-window* t)
            (:= *tr.space-mode* *tr.delayed-space-mode*)
            (:= *tr.window*     *tr.delayed-window*) ) )

        (tr.pick-trace) )              ;;; Build the next trace.

    (while *tr.trace-mis*)             ;;; While there are uncompacted
                                       ;;; ops left.
    (do
        (tr.display-trace-info *tr.trace-number*)

        (if (if (consp *tr.break-before-scheduling*)
                (member *tr.trace-number* *tr.break-before-scheduling*)
                *tr.break-before-scheduling*)
        (then
            (break-point before-scheduling) ) )

        (com.schedule)                 ;;; Schedule the trace.
        (tr.display-one-schedule)

        (tr.bookkeep)                  ;;; Add new ops wherever
                                       ;;; necessary.
        (if (if (consp *tr.break-after-bookkeeping*)
                (member *tr.trace-number* *tr.break-after-bookkeeping*)
                *tr.break-after-bookkeeping*)
        (then
            (break-point after-bookkeeping) ) )
        () )
    (result
        (tr.display-whole-sched-info *tr.trace-number*) ) )

    *tr.s*) )


(defun set-compactor-flags    ()

    (assert (member *tr.display-level*      '(0 1 2 3 4 5) ) )
    (assert (memq   *tr.space-mode*          '(nil mrt nsc cjo) ) )
    (assert (memq   *tr.delayed-space-mode*  '(nil mrt nsc cjo) ) )
    (assert (member *tr.trace-picker*        '(normal bb liberal) ) )
    () )


(defun com.schedule ()
    (let*( (first-mi     (car *tr.trace-mis*) )
           (live-before (mi:live-in first-mi) )
           (last-mi     (last-elt *tr.trace-mis*) )
           (live-after
```

```
            (mi:live-out-on-edge last-mi (mi:on-trace-direction last-mi)))
        (trace
            (for (mi in *tr.trace-mis*) (save
                `(,(car (mi:source mi) )
                  ,(mi:on-trace-direction mi)
                  ,mi
                  ,(if (mi:cond-jump? mi)
                       (mi:live-out-on-edge mi (mi:off-trace-direction mi))
                       () ) ) ) ) ) )

    (tr.display-generate-code-arguments live-before trace live-after)
    (:= *tr.schedule* (generate-code live-before trace live-after) )

    (:= *tr.schedule-size* (schedule:length *tr.schedule*) )
    (:= *tr.total-cycles* (+ *tr.total-cycles* *tr.schedule-size*) )
    () ) )
```

```
;==========================================================================
;
; COMPACT OPTIONS
;
; This module contains the definitions of options dealing with the trace
; compactor.
;
;==========================================================================

(eval-when (compile)
    (build '(utilities:options) ) )

(def-option *tr.display-level* 4 trace: "
Controls how much is printed out during compaction.
    0 - absolutely nothing.
    1 - only final stats.
    2 - like 1, but dumps a histogram of uncompacted MIs left during
        scheduling.  What fun.
    3 - trace info for each trace, no schedule.
    4 - trace info + schedule for each trace.
    5 - All of 4, plus the elements of each trace.
")


(def-option *tr.space-mode* () trace: "
Controls the way that space saving is done.
    ()   no space saving.
    'MRT minimum release times.  Won't let jumps get scheduled until the
         source order preceeding ops could have been scheduled, had there
         been no resource conflicts.
    'NSC no splice copies.  Refuses to schedule jumps until all the source
         order preceeding ops HAVE been scheduled.  Thus no splice copies
         could ever be generated.
    'CJO preserve the source order of conditional jumps by not allowing
         a conditional to move above a previous conditional.
")


(def-option *tr.window* () trace: "
When *TR.SPACE-MODE* = 'MRT is used, *TR.WINDOW* is subtracted from the
calculated release time.  Yes, Virginia, you can set *TR.WINDOW* to -100
and get a lot of empty cycles.
")


(def-option *tr.delayed-space-mode* () trace: "
If *TR.DELAYED-SPACE-MODE* is non-(), then starting with trace
*TR.DELAYED-SPACE-MODE-TRACE*, the *TR.DELAYED-SPACE-MODE* and
*TR.DELAYED-WINDOW* replace *TR.SPACE-MODE* and *TR.WINDOW*.
")


(def-option *tr.delayed-window* () trace: "
See *TR.DELAYED-SPACE-MODE*.
")


(def-option *tr.delayed-space-mode-trace* 2 trace: "
See *TR.DELAYED-SPACE-MODE*.
")
```



```
(def-option *tr.trace-picker* 'normal trace: "
Controls the way that traces are picked.
    'NORMAL  conservatively goes past splits and joins.
    'BB      basic block compaction only.
    'LIBERAL goes past splits and joins whenever it can,
             ignoring edge-probability and just using expect.
")


(def-option *tr.generate-trace-info?* () trace: "
Dumps certain trace-information into the output parallel NADDR.
    () do nothing.
    T  put (TRACE i) in each compacted instruction.
")



(def-option *tr.generate-code-hook* () trace: "
Undocumented, sorry.
")


(def-option *tr.dag-hook* () trace: "
Undocumented, sorry.
")


(def-option *tr.break-before-scheduling* () trace: "
If T then the compactor will stop at a breakpoint right before the current
trace is scheduled (given to the codegenerator).  If it is a list of trace
numbers, then the compactor will stop only on those traces.
")


(def-option *tr.break-after-bookkeeping* () trace: "
If T then the compactor will stop at a breakpoint right after the current
trace has been compacted and munged by the bookkeeper (i.e. right before
picking the next trace.  If it is a list of trace numbers, then the
compactor will stop only on those traces.
")

(def-option *tr.print-copying?* () trace: "
If T then the bookkeeper will print out information about all join and
split copies as they are made.
")
```

```
;===========================================================================
;
; Compiler Declarations.
;
; Any module that needs these declarations must do:
;
;     (INCLUDE TRACE:DECLARATIONS)
;
;===========================================================================

(eval-when (compile)
    (build '(
        utilities:sharp-sharp
        trace:mi
        interpreter:naddr
        ) ) )

(declare (special
    *tr.break-before-scheduling*
    *tr.break-after-bookkeeping*
    *tr.dag-hook*
    *tr.delayed-space-mode*
    *tr.delayed-window*
    *tr.delayed-space-mode-trace*
    *tr.display-level*
    *tr.generate-code-hook*
    *tr.generate-trace-info?*
    *tr.mi-number*
    *tr.ops-left*
    *tr.partial-rejoin-count*
    *tr.partial-rejoin-total*
    *tr.partial-split-count*
    *tr.partial-split-total*
    *tr.print-copying?*
    *tr.rejoin-count*
    *tr.rejoin-total*
    *tr.schedule-size*
    *tr.schedule*
    *tr.space-mode*
    *tr.split-count*
    *tr.split-total*
    *tr.start-length*
    *tr.s*
    *tr.total-cycles*
    *tr.trace-number*
    *tr.trace-picker*
    *tr.trace-mis*
    *tr.trace-size*
    *tr.unknown-copy-live-out*
    *tr.window*
    ) )
```

```
(eval-when (compile load)
    (include trace:declarations) )



(defun tr.display-trace-info ( traceno )

(caseq *tr.display-level*

    (2
        (if (= traceno 1) (then
            (msg 0 t "Histogram of total MIs left (* = 10 MIs):") ) )

        (msg 0)
        (loop (incr i from 1 to *tr.ops-left* by 10) (do
            (princ '*) ) ) )

    ( (3 4 5)
        (msg 0 t "Trace No.: " traceno "  Trace size: " *tr.trace-size*
            "  Offtrace OPs left: " *tr.ops-left* t)
        (msg "Trace elements: " (h (mi-list:numbers *tr.trace-mis*)
                                    10000 10000) )
        (msg 0 "Rejoin//split copies from last trace:          "
            *tr.rejoin-count* "//" *tr.split-count* t)
        (msg 0 "Partial-rejoin//split copies from last trace:  "
            *tr.partial-rejoin-count* "//" *tr.partial-split-count* t) ) )
() )


(defun tr.display-generate-code-arguments
    ( live-before source-record-list live-after )

    (if (== 5 *tr.display-level*) (then
        (msg 0 t "Arguments passed GENERATE-CODE:" t
            "LIVE-BEFORE = " (h live-before 100 100) t
            "LIVE-AFTER  = " (h live-after  100 100) t)
        (loop (for (oper direction mi off-live) in source-record-list)
             (incr i from 1)
        (do
            (msg (j i 3) "//" (j (mi:number mi) 4) ": "
                 (h oper 100 100) (t 35) " "
                 (j direction -7) " "
                 (j (mi:trace mi) 3) " "
                 (h off-live 100 100) t) ) )
        (msg t) ) ) )



(defun tr.display-one-schedule ()

(caseq *tr.display-level*
    ( (4 5)
        (msg 0 "Schedule:" t)

        (loop (incr cycleno from 1 to *tr.schedule-size*) (do
            (msg 0 t (j cycleno 3 #/-) "--")

            (for ( (oper source-mi) in (schedule:[] *tr.schedule* cycleno) )
            (do
                (if (> (flatsize oper) (chrct) ) (then
                    (msg 0 "      ") ) )
```



```
                (msg oper) ) )

            (msg t) ) ) ) )
() )



(defun tr.display-whole-sched-info  ( traceno )

(caseq *tr.display-level*
    ( (1 2 3 4 5)
        (msg 0 t t
            "FINAL SCHEDULING TOTALS: " (c (daytime) ) t
            "Length of original sequence:" (t 35) *tr.start-length* t
            "Total rejoin copies:"         (t 35) *tr.rejoin-total* t
            "Total split copies:"          (t 35) *tr.split-total*  t
            "Total partial-rejoin copies:" (t 35) *tr.partial-rejoin-total* t
            "Total partial-split copies:"  (t 35) *tr.partial-split-total*  t
            "Total traces:"                (t 35) (+ -1 traceno)          t
            "Total cycles scheduled:"      (t 35) *tr.total-cycles*       t)))
() )
```

```
;===========================================================================
;
; MI
;
; An MI (micro-instruction) represents either an uncompacted source operation
; or else a compacted instruction.   Taken as a group the MIs form the
; flow graph used by the trace scheduler.
;
;===================;
                    ;
(def-struct mi      ;
                    ;
;===================;
;
; The following fields are filled in when a record is made from a source
; instruction and handed to the compactor in the first place, and when
; a new record is made from old ones.
;
;===================;
                    ;
    number          ; A unique number used for naming this MI.
                    ;
    source          ; The list of the single source NADDR operation this
                    ; represents (if uncompacted) or the list of machine
                    ; operations in the instruction (if compacted).
                    ;
    expect          ; The relative probability of the MI's execution.
                    ;
    (succs          ; A list of the MIs to which control could next
      () suppressed); flow from this one.   SU is for successor.
                    ;
    (preds          ; A list of the MIs from which control could have flowed
        () suprress); to this one.  PR is for predecessor.
                    ;
    edge-prob       ; An assoc list.  Having (m4 .6) on m2's list means
                    ; that m4 is a successor with probability .6 of being
                    ; jumped to next after the execution of m4.  All of
                    ; the probs on the list should total 1, though that's
                    ; not necessary for the code to work correctly.
                    ; Functions edge-prob and edge-prob:set manipulate
                    ; the list.
                    ;
    compacted?      ; True if the MI represents a compacted machine
                    ; instruction or if the trace-scheduler no longer
                    ; wants to consider this record for scheduling.  False
                    ; if if this represents an uncompacted source
                    ; operation.
;===================;
;
; The following depend upon the choice of a trace and are filled in after
; an operation has been placed on a trace:
;
;===================;
                    ;
    (trace-succ     ; The next element on the current trace.
        () suppressed)
                    ;
    (trace-pred     ; The previous element on the current trace.
        () suppressed)
                    ;
    trace-pos       ; Which element (1-based) of the current trace this is.
                    ;
    first-cycle     ; First and last cycles of the generated machine
```

1

PS:<C.S.BULLDOG.TRACE>MI.LSP.1

```
    last-cycle      ; machine operations corresponding to this source
                    ; operation.
                    ;
    trace-direction ; Used in an MI built from several MIs compacted
                    ; together.  Used when n > 0 cond-jumps are compacted
                    ; to form part of the MI.  A list containing n
                    ; elements, each LEFT or RIGHT and each corresponding
                    ; to the succs in the same position in the list
                    ; succs (which is itself, however, of length n+1).
                    ; Also corresponds to the cond-jumps in the MI:SOURCE
                    ; field, which is expected to have its cond-jumps
                    ; in source order.  A LEFT in the field indicates
                    ; that the jump is to the next trace element if true,
                    ; off-trace if false.  RIGHT is the reverse.
                    ;
;===================;
;
; The following fields are used during bookkeeping:
;
;===================;
                    ;
    (constituents   ; As a new MI is being formed out of a list of ops
                    ; which were scheduled in the same cycle, this field
       () suprress); buffers the list of those ops.
                    ;
    copy-type       ; For uncompacted MIs only, one of:
                    ;   ()    - if this MI is an original source instructions.
                    ;   JOIN  - if this MI is a rejoin copy.
                    ;   SPLIT - if this MI is a split copy.
                    ;
    (copy-live-out
       *tr.unknown-copy-live-out*)
                    ; For uncompacted rejoin or split copy MIs only; this
                    ; contains the variables live on exit from the MI.
                    ; For original source MIs, we get the live info from
                    ; the flow analysis.
                    ;
    trace           ; This field is used for picture drawing and debugging
                    ; only.  It is the number of the trace that produced
                    ; this MI.  If the MI is compacted, then it is the
                    ; number of the trace that compacted it; if
                    ; uncompacted, it is the trace made it or () for
                    ; original source MIs.
                    ;
;===================;
;
; The following is used during the translation of MIs to NADDR.
;
;===================;
                    ;
    translated-to-source
    )               ; Contains the actual pnaddr produced from the MI.
;===================;
;
;
; (MI:OPERATOR MI)
;     For uncompacted MIs, the NADDR operator.  () for compacted MIs.
;
; (MI:OPER MI)
;     For uncompacted MIs, the NADDR operation.  () for compacted MIs.
;
; (MI:COND-JUMP? MI)
;     True if MI represents a conditional jump (has more than one successor).
```

2

```
;
; (MI:REJOIN? MI)
;     True if MI is jumped to by other MIs (has more than one predecessor).
;
; (MI-LIST:NUMBERS MI-LIST)
;     Takes a list of MIs and returns a corresponding list of their numbers.
;
; ##MI 36
;     Returns the MI numbered 36.
;
; (MI:LIVE-OUT MI)
;     Returns the list of names live on exit from MI.
;
; (MI:LIVE-IN MI)
;     Returns the list of names live on entrance to MI.
;
; (MI:LIVE-OUT-ON-EDGE MI DIRECTION)
;     Returns the list of variables live on entrance to one of MI's
;     successors.  DIRECTION is LEFT or RIGHT or (), selecting either
;     the left or right successor of MI or neither successor.
;
; (MI:ON-TRACE-DIRECTION  MI)
; (MI:OFF-TRACE-DIRECTION MI)
;     These functions return which way (LEFT or RIGHT) the on-trace and
;     off-trace edges of an MI go (LEFT always for a non-conditional-jump).
;     If MI has no on-trace successor marked yet, the trace is assumed
;     to go to the left.
;
; (MI:OFF-TRACE-SUCC MI)
;     The off-trace successor of MI.   Valid only during bookkeeping.
;
; (MI:OFF-TRACE-PREDS MI)
;     The list of off-trace predecessors of MI.  Valid only during
;     bookkeeping.
;
;=========================================================================

(eval-when (compile)
    (build '(
        utilities:sharp-sharp
        interpreter:naddr) ) )

(defvar *tr.unknown-copy-live-out* (cons '*tr.unknown-copy-live-out*) )
    ;
    ;*** Special marker showing that we don't know the value of
    ;*** MI:COPY-LIVE-OUT field.


(defun mi:operator ( mi )
    (assert (mi:is mi) )
    (if (mi:compacted? mi)
        ()
        (oper:operator (car (mi:source mi) ) ) ) )

(defun mi:oper ( mi )
    (assert (mi:is mi) )
    (if (mi:compacted? mi)
        ()
        (car (mi:source mi) ) ) )

(defun mi:cond-jump? ( mi )
    (assert (mi:is mi) )
    (> (length (mi:succs mi) ) 1) )
```



```
(defun  mi:rejoin? ( mi )
    (assert (mi:is mi) )
    (> (length (mi:preds mi) ) 1) )


(defun mi-list:numbers ( mi-list )
    (for (mi in mi-list) (save
        (assert (mi:is mi) )
        (mi:number mi) ) ) )



(def-sharp-sharp mi
    `(mi-with-number ,(read) ) )

(declare (special *tr.s*) )
(defun mi-with-number (number)
    (loop (for elt in *tr.s*) (do
        (if (= (mi:number elt) number) (then
            (return elt) ) ) )
    (result nil) ) )


(defun mi:live-out ( mi )
    (assert (&& (mi:is mi)
                (! (mi:compacted? mi) ) ) )
    (if (== *tr.unknown-copy-live-out* (mi:copy-live-out mi) )
        (oper:live-out (car (mi:source mi) ) )
        (mi:copy-live-out mi) ) )


(defun mi:live-in ( mi )
    (assert (&& (mi:is mi)
                (|| (! (mi:compacted? mi) )
                    (! (mi:source     mi) ) ) ) ) )
        ;
        ;*** MI is either uncompacted or else it is a dummy MI.

    (if (== *tr.unknown-copy-live-out* (mi:copy-live-out mi) ) (then
        (oper:live-in (car (mi:source mi) ) ) )

    (else
        (unionq
            (top-level-removeq (oper:part (car (mi:source mi) ) 'written)
                               (mi:copy-live-out mi) )
            (loop (for-each-oper-operand-read (car (mi:source mi) ) name)
                  (save name) ) ) ) ) )


(defun mi:live-out-on-edge ( mi direction )
    (assert (&& (mi:is mi)
                (! (mi:compacted? mi) )
                (memq direction '(left right) ) ) )

    (? ( (== *tr.unknown-copy-live-out* (mi:copy-live-out mi) )
         (oper:live-out-on-edge (car (mi:source mi) ) direction) )

       ( (! (mi:cond-jump? mi) )
         (mi:copy-live-out mi) )

       ( t
         (intersectionq
```

```
            (mi:copy-live-out mi)
            (mi:live-in
                (caseq direction
                    (left  (car  (mi:succs mi) ) )
                    (right (cadr (mi:succs mi) ) ) ) ) ) ) ) )


(defun mi:on-trace-direction ( mi )
    (assert (mi:is mi) )
    (if (|| (! (mi:trace-succ mi) )
            (== (car (mi:succs mi) )  (mi:trace-succ mi) ) )
        'left
        'right) )

(defun mi:off-trace-direction ( mi )
    (assert (mi:is mi) )
    (assert (== 2 (length (mi:succs mi) ) ) )

    (if (|| (! (mi:trace-succ mi) )
            (== (car (mi:succs mi) )  (mi:trace-succ mi) ) )
        'right
        'left) )


(defun mi:off-trace-succ ( mi )
    (assert (mi:is mi) )
    (if (== (mi:trace-succ mi) (car (mi:succs mi) ) )
        (cadr (mi:succs mi) )
        (car  (mi:succs mi) ) ) )


(defun mi:off-trace-preds ( mi )
    (assert (mi:is mi) )
    (top-level-removeq (mi:trace-pred mi) (mi:preds mi) ) )
```

```
;==========================================================================
;
; This module convert compacted microinstructions into parallel naddr code.
;
; To convert a list of mi's (such as returned by the compacter) into
; parallel naddr (PNADDR):
;
;
;       (mis->pnaddr mi-list)
;
; The conversion is very simple minded-- preorder traversal of the
; flow graph, starting at the one node that has the (START) source.
; Each node is visited only once (using the MI:TRANSLATED-TO-SOURCE
; flag to remember visits).
;
; It is assumed that there are one or two flow successor for each MI.
; Conditional boolean jumps (TRUEGO, FALSEGO) are converted into the
; parallel naddr form with two explicit labels, like the if-then-else
; naddr operations.
;
;==========================================================================

(eval-when (compile load)
    (include trace:declarations) )


(declare (special
    *mis-to-do*          ;*** stack of mi's possibly not yet processed
    ) )



;***
;*** Translate the list of mi's, MI-LIST, into a a parallel naddr program.
;***

(defun mis->pnaddr ( mi-list )
    (assert (listp mi-list) )
    (:= *mis-to-do* () )

    (let ( (pnaddr-stream () )
           (pnaddr        () ) )

            ;*** clear the "translated" flag of every mi, and if it has
            ;*** no predecessors, push it on our to-do stack.
            ;
        (for (mi in mi-list)
             (do (:= (mi:translated-to-source mi) () )
                 (if (== 'def-block (caar (mi:source mi) ) )
                     (push *mis-to-do* mi) ) ) )

            ;*** while there are untranslated mi's, do
            ;***     convert each one to pnaddr
            ;
        (loop
            (initial mi () )
            (while *mis-to-do*)
            (do (pop *mis-to-do* mi)
                (if (! (mi:translated-to-source mi) ) (then
                    (if (:= pnaddr (mi:pnaddr mi) )
                        (push pnaddr-stream pnaddr) ) ) ) ) )

        (dreverse pnaddr-stream) ) )
```



```
;***
;*** Translate the single mi MI into a parallel naddr statement.  As a
;*** side effect, push on *MIS-TO-DO* the successors of this mi.  Only
;*** the labels of source naddr statements are changed (for jumps).
;*** The new instruction is also placed in the translated-to-source
;*** field of the mi.

(defun mi:pnaddr ( mi )
    (assert (mi:is mi) )

    (:= (mi:translated-to-source mi) t)

    (let ( (popers      () )
           (cond-oper   () )
           (trace-dir   (mi:trace-direction mi) )
           (succs       (mi:succs mi) ) )

                ;*** generate a label for this MI if this isn't a DEF-BLOCK.
                ;
        (caseq (oper:group (car (mi:source mi) ) )
          ( (def-block) )
          ( t
            (push popers '(label ,(mi:pnaddr-label mi) ) ) ) ) )

                ;*** translate each of the source naddr statements
                ;
        (for (oper in (mi:source mi) )
             (do (caseq (oper:group oper)
                   (goto)

                   (def-block
                       (:= popers (reverse oper) ) )    ;*** sigh, hack

                   (cond-jump
                       (push cond-oper
                             '( ,(oper:operator oper)
                                ,(oper:part oper 'read1)
                                ,(oper:part oper 'read2)
                                ,(oper:part oper 'probability)
                                ,(if (== (car trace-dir) 'right)
                                     (then (mi:pnaddr-label (car  succs) )))
                                ,(if (== (car trace-dir) 'left)
                                     (then (mi:pnaddr-label (car  succs) )))))

                       (pop succs)
                       (pop trace-dir) )

                   (if-then-else
                       (push cond-oper
                             '( ,(oper:operator oper)
                                ,(oper:part oper 'read1)
                                ,(oper:part oper 'read2)
                                ,(oper:part oper 'probability)
                                ,(if (== (car trace-dir) 'right)
                                     (then (mi:pnaddr-label (car  succs) )))
                                ,(if (== (car trace-dir) 'left)
                                     (then (mi:pnaddr-label (car  succs) )))))

                       (pop succs)
                       (pop trace-dir) )
```

```
                  (t
                       (push popers oper) ) ) ) )

              ;*** record the conditional jump, if any,  as a COND
              ;
        (if cond-oper (then
           (push cond-oper '(goto ,(mi:pnaddr-label (car  succs) ) ) )
           (push popers '(? ,(dreverse cond-oper) ) ) ) )

              ;*** push the successors on the stack
              ;
        (:= succs (mi:succs mi) )       ;*** in case you forgot (ha ha).
        (for (succ in succs)
             (do (if (! (mi:translated-to-source succ) )
                       (push *mis-to-do* succ) ) ) )

              ;*** Hack, if there is one successor and it's already
              ;*** been translated, we need to do an explicit GOTO.
              ;*** O.w.  it will be done next, it's code will follow
              ;*** immediately.
              ;
        (if (&& (= 1 (length succs) )
                (mi:translated-to-source (car succs) ) )
            (push popers '(goto ,(mi:pnaddr-label (car succs) ) ) ) )

        (:= (mi:translated-to-source mi)  (dreverse popers))
        ) )



;***
;*** Return the label of an MI.
;***

(defun mi:pnaddr-label ( mi )
    (assert (mi:is mi) )
    (atomconcat 'l (mi:number mi) ) )
```

```
;;; N-ADDRESS CODE INTO COMPACTION RECORDS TRANSLATION.          J. Fisher
;;;                                                              11/30/81


(eval-when (compile load)
    (include trace:declarations) )



(declare (special
    *nr.xref-list*
    ) )

(defun tr.get-*tr.s* ( naddr )
      (nr.naddr-to-records naddr) )


;;; nr.naddr-to-records does the actual conversion.  First, an end statement
;;; is provided by nr.cleanly-end-program if none exists.  Then records are
;;; formed, but these have actual names rather than pointers where other
;;; records belong (i.e. as flow successors) and have names, not the needed
;;; numbers, for registers.  A final routine fixes that...

(defun nr.naddr-to-records  (naddr-list)
    (nr.form-records  (nr.cleanly-end-program naddr-list))
    (nr.replace-names-with-pointers)
    (nr.build-flow-prs)
    (nr.eliminate-gotos)
    (nr.set-all-edge-probabilities)
    (nr.calculate-expects)

    (:= *nr.xref-list* ())
    nil)


;;; To form records, we  the following.  The list is processed statement
;;; by statement.  If the statement is not a label or an expect, a record is
;;; built by a call to nr.make-new-record.  Then all the labels that referred
;;; to this statement are placed in the label table as doing so.
;;; When a label is encountered, it is placed on the currently active list,
;;; where labels are accumulated until an operative statement is encountered.
;;;
;;; SPECIAL CASES:
;;;      - Unknown opcodes are ignored.
;;;      - Little error checking is done, but missing labels are reported.


(defun nr.form-records (naddr-list)

    (:= *tr.s* nil)
    (:= *nr.xref-list* nil)

    (let ((active-labels nil)
          (stat-count 0)
          (current-record nil)   )

         (for (stat in naddr-list)
              (do
               (cond
                ((nr.type-of-stat-that-makes-records stat)
                     (++ stat-count)
                     (:= current-record (nr.make-new-record
```



```
                                           stat  stat-count))
                (push *tr.s* current-record)
                (nr.put-xref stat-count current-record)
                (for (lab in active-labels)
                     (do (nr.put-xref lab current-record)))
                (:= active-labels nil))
            ((== 'label (oper:operator stat) )
                (push active-labels (oper:part stat 'label1) ))))))
    nil)


;;; An end statement is provided if none exists.  Thus problems like end
;;; labels and fall throughs are OK.

(defun nr.cleanly-end-program  (naddr-list)
    (if (not (equal (last-elt naddr-list) '(end)  ))
        (then (nconc naddr-list '( (end) )    ))
        (else naddr-list)                        ))


;;; In mi:succs of each formed record, we have a name for each follower.
;;; This is changed to be the actual pointer.

(defun nr.replace-names-with-pointers  ()
    (for (elt in *tr.s*)
         (do
          (:= (mi:succs elt)   (mapcar 'nr.get-xref (mi:succs elt)))
                                                                    )))

;;; Little self-evident syntactic sugar utilities:

(defun nr.type-of-stat-that-makes-records (stat)
    (memq (oper:group stat)
          '(two-in-one-out one-in-one-out vload vstore if-then-else
            goto esc live end stop trace-fence loop-end loop-start
            def-block dcl assert use def loop-assign) ) )


;;; Self evident storage routines to cross-reference names of registers vs.
;;; numbers, and labels vs. statements.

(defun nr.put-xref (lab mi)
    (push *nr.xref-list* (list lab mi) ))

(defun nr.get-xref (lab)
    (let ((result (cadr (assoc# lab *nr.xref-list*))))
         (if result result
             (else (error (list lab "NR.GET-XREF: Missing label.") ) ) ) ) )



;;; nr.make-new-record produces a record out of a single naddr statement.  The
;;; fields are default set to the values that work for the two-in and one-in
;;; two-out cases, and are changed where necessary for the other cases.

(defun nr.make-new-record (stat number)

(let* ( (name          number)
        (fall-through (+ 1 number) )
        (mi            (mi:new
                         source      (list stat)
                         number      (++ *tr.mi-number*)
                         succs    (list fall-through) ) ) )
```

```
        ;;; change default field values for special operators
    (caseq (oper:group stat)

        (cond-jump
            (:= (mi:succs mi)
                `(,(oper:part stat 'label1)
                  ,(|| (oper:part stat 'label2)
                       fall-through) ) ) )

        (if-then-else
            (:= (mi:succs mi)
                `(,(oper:part stat 'label1)
                  ,(|| (oper:part stat 'label2)
                       fall-through) ) ) )

        (goto
            (:= (mi:succs mi) (list (cadr stat) ) ) )

        ( (live def-block esc)
            (:= (mi:compacted? mi) t) )

        ( (end stop)
            (:= (mi:succs  mi) () )
            (:= (mi:compacted? mi) t) ) )
    mi) )



;;; We've built only flowsus, so here we build the
;;; set of flowprs...

(defun nr.build-flow-prs ()
            (for (m in *tr.s*)
                 (do
                  (for (succ in (mi:succs m))
                       (do
                        (push (mi:preds succ) m))))))



;;; THE FOLLOWING IS THE STUFF RELATED TO FILLING IN THE EXPECT AND JUMP
;;; JUMP PROBABILITY FIELDS


;;;  He's making a list (ta ta), checking it twice (la la).

(defun nr.calculate-expects  ()
    (for (elt in (reverse *tr.s*))
         (do (mi:find-expect elt))))


;;; mi:find-expect retrieves the mi:expect value of an MI, or forces
;;; a calculation of it if it hasn't been done yet.  Because of possible
;;; flakiness in various codegenerators, we guarrantee the expect is at
;;; least a tiny number.

(defun mi:find-expect ( elt )

    (if (! (mi:expect elt) ) (then

        (if (! (mi:preds elt) ) (then
```



```
            (:= (mi:expect elt) 1.0) )
        (else
            (:= (mi:expect elt) 1.0)        ;;; protect against circularity
            (:= (mi:expect elt)
                (loop (for pred in (mi:preds elt) )
                      (when (! (trace-fence:is pred) ) )
                      (initial expect 0.0)
                (do
                    (:= expect (+ expect
                        (if (mi:cond-jump? pred)
                            (* (mi:find-expect pred)
                               (nr.edge-prob pred elt) )
                            (mi:find-expect pred) ) ) ) )
                (result expect) ) ) ) )

        (if (loop-start:is elt) (then
            (:= (mi:expect elt)
                (* (mi:expect elt) (mi:iteration-count elt) ) ) )

        (else (if (loop-end:is elt) (then
            (let ( (loop-start (nr.find-loop-start (loop-end:name elt) ) ) )
                (:= (mi:expect elt)
                    (// (mi:find-expect loop-start)
                        (mi:iteration-count loop-start) ) ) ) ) ) ) )

        (:= (mi:expect elt) (max 1.0E-30 (mi:expect elt) ) ) ) )

    (mi:expect elt) )


(defun mi:iteration-count (elt)
    (caddar (mi:source elt)))


(defun nr.find-loop-start  (name)

    (loop
     (for elt in *tr.s*)
     (do
      (if (&&
            (loop-start:is elt)
            (eq (loop-start:name elt) name))
          (return elt)))
     (result nil)))


(defun loop-start:is (elt)
    (== 'loop-start (oper:operator (car (mi:source elt) ) ) ) )

(defun trace-fence:is (elt)
    (== 'trace-fence (oper:operator (car (mi:source elt) ) ) ) )

(defun loop-start:name (elt)
    (oper:part (car (mi:source elt) ) 'label1) )

(defun loop-end:is (elt)
    (== 'loop-end (oper:operator (car (mi:source elt) ) ) ) )

(defun loop-end:name (elt)
    (oper:part (car (mi:source elt) ) 'label1) )
```

```
;;; Edge prob is formed here, and passed along when copies are made in the
;;; bookkeeping phase.

(defun nr.edge-prob  (pred succ)

    (assert (memq succ (mi:succs pred)))

    (loop
     (for next-succ in (mi:succs pred))
     (for next-edge-prob in (mi:edge-prob pred))
     (do (if (== succ next-succ)
             (then (return next-edge-prob))))))


(defun nr.set-all-edge-probabilities ()

    (for (elt in *tr.s*)
         (do
          (if (mi:cond-jump? elt)
              (then
               (:= (mi:edge-prob elt)
                   '(,(mi-jump:prob elt) ,(- 1.0 (mi-jump:prob elt)))))
              (else
               (:= (mi:edge-prob elt) '(1.0)))))))


(defun mi-jump:prob (elt)
    (oper:part (car (mi:source elt)) 'probability))


;;;*************************************************************************
;;;
;;; Here we attempt to eliminate all the goto statements.  It is remarkably
;;; easy if this really works... It is only difficult in that we are altering
;;; *tr.s* at the same time that we're running through it.
;;;

(defun nr.eliminate-gotos ()

    (loop
     (initial goto-mi (nr.find-goto-mi) )
     (while goto-mi)
     (do (bk.unsplice goto-mi)
         (:= *tr.s* (set-diffq *tr.s* (list goto-mi))) )
     (next goto-mi (nr.find-goto-mi) ) ))


(defun nr.find-goto-mi ()
    (loop
     (for elt in *tr.s*)
     (do
      (if (eq (oper:group (car (mi:source elt)))  'goto)
          (return elt)))
     (result nil)))
```

```
;===========================================================================
;
; TRACE PICKER
;
; This module implements the picking of the next trace of uncompacted MIs from
; the MI flow graph.
;
; (TR.INITIALIZE-TRACE-PICKER)
;     Initializes the module by clearing *TR.TRACE-MIS*.
;
; (TR.PICK-TRACE)
;     Picks the next trace from the flow graph, setting *TR.TRACE-MIS* to
;     be the trace and *TR.TRACE-SIZE* the size of the trace and
;     decrementing *TR.OPS-LEFT*.   *TR.TRACE-MIS* is set to be () if there
;     are no uncompacted MIs left.
;
; Traces are picked by first finding the uncompacted MI with the largest
; :EXPECT value; that MI becomes the seed of the trace.  The trace picker
; then moves forward, incrementally growing the trace from the end.  To
; find the next MI, it looks at the current end of the trace and finds
; the successor that meets the trace criteria and adds it to the end of
; the trace.  After growing forward, the trace picker grows the trace
; backwards analogously.
;
; Whichever direction we are growing the trace, the same criteria are
; used to see if an edge between two MIs belongs to the trace.  Suppose
; we have two MIs, PRED-MI and SUCC-MI.  If we are growing the trace
; forward, PRED-MI is the current end of the trace.  If we are growing
; the trace backward, then SUCC-MI is the current beginning of the trace.
; In either case, we look at the edge from PRED-MI to SUCC-MI; both of
; the MIs must be uncompacted.
;
; The setting of *TR.TRACE-PICKER* determines which edge criteria are used:
;
;     NORMAL
;         The edge from PRED-MI to SUCC-MI has the highest edge probability
;         of all exits from PRED-MI.
;
;         The edge from PRED-MI to SUCC-MI is the most likely to be
;         executed of all the predecessor edges coming into SUCC-MI.  That
;         is, the edge contributes the most :EXPECT to SUCC-MI of all
;         of the predecessor edges.
;
;     BB
;         "Basic block" -- traces consist exactly of basic blocks.
;
;     LIBERAL
;         PRED-MI has the highest :EXPECT of all the predecessors of SUCC-MI.
;
;         SUCC-MI has the highest :EXPECT of all the successors of PRED-MI.
;
; In all three criteria, PRED-MI and SUCC-MI must be uncompacted.  To
; guarrantee that a TRACE-FENCE pseudo-op can only occur at the end of
; a trace, PRED-MI can't be a TRACE-FENCE pseudo-op.
;
;===========================================================================


(eval-when (compile load)
    (include trace:declarations) )


(defun tr.initialize-trace-picker ()
```

1

PS:<C.S.BULLDOG.TRACE>TRACE-PICKER.LSP.1

```
    (:= *tr.trace-mis* () )
    () )


(defun tr.pick-trace ()
    (let ( (seed-mi (tp.mi-list:max-expect-mi *tr.s*) ) )

        (:= *tr.trace-mis* () )

            ;*** Gobble MIs forward from the seed, marking the :TRACE-POS
            ;*** field of picked MIs so that we won't pick them again.
            ;
        (loop (initial next-mi seed-mi)
              (while next-mi)
        (do
            (:= (mi:trace-pos next-mi) t)
            (push *tr.trace-mis* next-mi) )
        (next next-mi (tp.mi:next-forward-trace-mi next-mi) ) )

            ;*** Reverse the trace and remove the seed from the front.
            ;
        (:= *tr.trace-mis* (dreverse &&&) )
        (pop *tr.trace-mis*)

            ;*** Gobble MIs backward from the seed.
            ;
        (loop (initial next-mi seed-mi)
              (while next-mi)
        (do
            (:= (mi:trace-pos next-mi) t)
            (push *tr.trace-mis* next-mi) )
        (next next-mi (tp.mi:next-backward-trace-mi next-mi) ) )

            ;*** Set the trace size and number of operations left.
            ;
        (:= *tr.trace-size* (length *tr.trace-mis*) )
        (:= *tr.ops-left*   (- *tr.ops-left* *tr.trace-size*) )
        () ) )


;***=======================================================================
;***
;*** (TP.MI:NEXT-FORWARD-TRACE-MI MI)
;***
;*** MI is assumed to be already on the trace.  Returns the successor
;*** of MI that should be on the trace.  If there is no appropriate
;*** successor, () is returned.
;***
;***=======================================================================

(defun tp.mi:next-forward-trace-mi ( mi )
    (loop (for succ-mi in (mi:succs mi) ) (do
        (if (&& (! (mi:trace-pos succ-mi) )
                (tp.pred-mi:succ-mi:right-edge? mi succ-mi) )
        (then
            (return succ-mi) ) ) )
    (result () ) ) )


;***=======================================================================
;***
;*** (TP.MI:NEXT-BACKWARD-TRACE-MI MI)
;***
```

2

```
;*** MI is assumed to be already on the trace.  Returns the predecessor
;*** of MI that should be on the trace.  If there is no appropriate
;*** successor, () is returned.
;***
;***==========================================================================

(defun tp.mi:next-backward-trace-mi ( mi )
    (loop (for pred-mi in (mi:preds mi) ) (do
        (if (&& (! (mi:trace-pos pred-mi) )
                (tp.pred-mi:succ-mi:right-edge? pred-mi mi) )
        (then
            (return pred-mi) ) ) )
    (result () ) ) )



;***==========================================================================
;***
;*** (TP.PRED-MI:SUCC-MI:RIGHT-EDGE? PRED-MI SUCC-MI)
;***
;*** One of PRED-MI and SUCC-MI are already on the trace.  Returns true
;*** if the edge from PRED-MI to SUCC-MI meets the trace picking criteria
;*** and thus should be part of the trace.
;***
;***==========================================================================

(defun tp.pred-mi:succ-mi:right-edge? ( pred-mi succ-mi )

    (if (|| (! pred-mi)
            (! succ-mi)
            (mi:compacted? pred-mi)
            (mi:compacted? succ-mi)
            (== 'trace-fence (mi:operator pred-mi) ) )
    (then
        nil)

    (else
        (caseq *tr.trace-picker*
            (normal
                (&& (>= (nr.edge-prob pred-mi succ-mi) .5)

                        ;** See if PRED-MI contributes the most of all
                        ;** of SUCC-MI'S predecessors
                        ;
                    (= (* (nr.edge-prob pred-mi succ-mi)
                          (mi:expect pred-mi) )

                       (loop (for next-pred-mi in (mi:preds succ-mi) )
                             (initial max-expect 0.0)
                             (bind expect (* (nr.edge-prob next-pred-mi
                                                           succ-mi)
                                             (mi:expect next-pred-mi) ) )
                       (do
                          (if (> expect max-expect) (then
                              (:= max-expect expect) ) ) )
                       (result max-expect) ) ) ) )

            (bb
                (&& (! (mi:cond-jump? pred-mi) )
                    (! (mi:rejoin?    succ-mi) ) ) )

            (liberal
                (&& (== succ-mi
```



```
                        (tp.mi-list:max-expect-mi (mi:succs pred-mi) ) )
                    (== pred-mi
                        (tp.mi-list:max-expect-mi (mi:preds succ-mi)))))))))


;***==========================================================================
;***
;*** (TP.MI-LIST:MAX-EXPECT-MI MI-LIST)
;***
;*** Returns the uncompacted MI in MI-LIST with the largest :EXPECT value.
;*** Returns () if there is no such uncompacted MI.
;***
;***==========================================================================

(defun tp.mi-list:max-expect-mi ( mi-list )
    (loop (for mi in mi-list)
          (initial max-expect     -1
                   max-elt-so-far () )
    (do
        (if (&& (! (mi:compacted? mi) )
                (> (mi:expect mi) max-expect) )
        (then
            (:= max-expect (mi:expect mi) )
            (:= max-elt-so-far mi) ) ) )
    (result max-elt-so-far) ) )
```