```
;***----------------------------------------------------------------------********
;*** Copyright (C) 1983 John R. Ellis.  This file may be used for      ********
;*** educational and research purposes only, by the faculty, students, ********
;*** and staff of Yale University only.                                ********
;***----------------------------------------------------------------------********
;===============================================================================
;
; BOTTOM-UP GREEDY
;
; (BUG.ASSIGN-MES)
;
; This module assigns MEs to each VN, using an algorithm inspired by the
; top-down greedy register allocation.  This algorithm is "bottom-up"
; because we draw our dags right (entrance at the top, exit at the bottom).
;
; The general algorithm is recursive -- choose functional units for the
; operands of a VN, then choose a functional unit for the VN based on
; those chosen for the operands.  A VN is assigned only after all its
; operands and constraining VNs are assigned.  When choosing functional
; units for the operands of a VN, we are "mindful" of the intended
; destination of the VN and the possible functional-units the VN can be
; done on.
;
; When recursing up through the DAG, the VNs with greatest depth are done
; first.   This is similar to the "highest levels first" strategy of
; list scheduling.
;
;===============================================================================


(eval-when (compile load)
    (include list-scheduler:declarations) )

(defun bug.assign-mes ()
    (heights-depths.assign)
    (resource.initialize-schedule 'bug)
    (registers.initialize)

    (bug.assign-likely-mes)

    (loop (for vn in (bug.depth-sort-vns *ls.exit-vns*) ) (do
        (bug.vn:assign vn () ) ) )

    (bug.make-defis&useis)
    () )


;***============================================================================
;***
;*** (BUG.ASSIGN-LIKELY-MES)
;***
;*** Initializes :LIKELY-MES of each VN to a list of feasible MEs that
;*** could be used for the VN.  For DEF and USE VNs, the list is set to
;*** the MEs of the locations of the DEF or USE (if given); if a DEF has
;*** no locations, but a USE that reads the DEF does, the DEF adopts the
;*** :LIKELY-MES of the USE.  For operations VNs, it is set to the list
;*** of functional-unit MEs that can compute the value of the VN.
;***
;***============================================================================

(defun bug.assign-likely-mes ()
    (loop (for-each-vn vn) (do
        (caseq (vn:type vn)
            (pseudo-op)
```

```
            ( (def use)
                (:= (vn:likely-mes vn)
                    (loop (for (me register) in (vn:locations vn) ) (save
                        me) ) ) )
            (copy
                (:= (vn:likely-mes vn) () ) )
            (operation
                (:= (vn:likely-mes vn)
                    (loop (for me in (operator:functional-unit-mes
                                        (vn:operator vn) ) )
                          (when (me:datum:ok? me vn) )
                    (save me) ) ) )
            (t
                (error (list vn "BUG.ASSIGN-LIKELY-MES: Case error."))))))

    (loop (for-each-vn vn)
          (when (&& (== 'def (vn:type vn) )
                    (! (vn:likely-mes vn) ) ) )
          (bind use-vn (car (for-some (reading-vn in (vn:reading-vns vn) )
                                (== 'use (vn:type reading-vn) ) ) ) )
          (when use-vn)
    (do
       (:= (vn:likely-mes vn) (vn:likely-mes use-vn) ) ) )

    () )


;***============================================================================
;***
;*** (BUG.VN:ASSIGN VN DESTINATIONS)
;***
;*** This is the recursive function that assigns a VN, setting :ME,
;*** :LIKELY-MES, and :BUG-CYCLE of VN to reflect our choice of where to
;*** compute VN.  DESTINATIONS is a list of MEs where we want the value
;*** of VN to end up in (the value need end up in only one of them); if
;*** DESTINATIONS is () that means we don't care where the value goes.
;***
;*** After assigning VN, all the operands of VN are examined; any DEF
;*** operands are reassigned based on what was chosen for VN.
;***
;***============================================================================

(defun bug.vn:assign ( vn destinations )
    (if (! (vn:bug-cycle vn) ) (then
        (caseq (vn:type vn)
            (pseudo-op)
            (def
                (bug.def-vn:assign       vn destinations) )
            (use
                (bug.use-vn:assign       vn destinations) )
            (copy
                (bug.copy-vn:assign      vn destinations) )
            (operation
                (bug.operation-vn:assign vn destinations)

                (loop (for operand-vn in (vn:operand-vns vn) )
                      (when (== 'def (vn:type operand-vn) ) )
                (do
                    (bug.def-vn:reassign operand-vn (vn:me vn) ) ) ) )

            (t
                (error (list vn "BUG.VN:ASSIGN: Case error.") ) ) ) ) )
    () )
```

```
;***===========================================================================
;***
;*** (BUG.USE-VN:ASSIGN VN DESTINATIONS)
;***
;*** Assigns a USE VN merely by recursively asssigning the operand of
;*** the USE.
;***
;***===========================================================================

(defun bug.use-vn:assign ( vn destinations )
    (bug.vn:assign (car (vn:operand-vns vn) )
                   (vn:likely-mes vn) )
    (:= (vn:bug-cycle vn) (vn:bug-cycle (car (vn:operand-vns vn) ) ) )
    () )


;***===========================================================================
;***
;*** (BUG.COPY-VN:ASSIGN VN DESTINATIONS)
;***
;*** Assigns a COPY VN by recursively assigning its operand.  It is assumed
;*** that the copy operation will take one cycle and can be done in
;*** whatever register bank contains the operand.
;***
;***===========================================================================

(defun bug.copy-vn:assign ( vn destinations )
    (loop (for operand-vn
               in (bug.depth-sort-vns (append (vn:constraining-vns vn)
                                              (vn:operand-vns      vn) ) ) )
    (do
        (bug.vn:assign operand-vn
                       (if (memq operand-vn (vn:operand-vns vn) )
                           destinations
                           () ) ) ) )

    (:= (vn:bug-cycle  vn) (+ 1 (bug.vn:available (car (vn:operand-vns vn) ) ) )
    (:= (vn:likely-mes vn) (vn:likely-mes (car (vn:operand-vns vn) ) ) )
    () )


;***===========================================================================
;***
;*** (BUG.DEF-VN:ASSIGN VN DESTINATIONS)
;***
;*** VN is a DEF.  We assume that the machine has the bandwidth to provide
;*** VN to its readers on cycle 0 (see the next function).
;***
;***===========================================================================

(defun bug.def-vn:assign ( vn destinations )
    (:= (vn:bug-cycle vn) 0) )


;***===========================================================================
;***
;*** (BUG.OPERATION-VN:ASSIGN VN DESTINATIONS)
;***
;*** VN is an operation VN.  Picks a good ME for for VN.  First all the
;*** predecessor VNs (operands and constraining) are recursively assigned,
;*** doing the ones of largest depth first.  As we assign each operand,
```

```
;*** we make a "best guess" about the MEs likely to be used for VN; that
;*** best guess is pased on as DESTINATIONS for the recursive assignment
;*** to the operand.  After we've done all the operands, we then make
;*** a final best guess for VN, preferring an ME that is closest to the
;*** operands if we have any choice left.
;***
;*** After we've picked an ME and a cycle, we schedule it on the resource
;*** schedule to indicate we've made a choice.
;***
;***===========================================================================

(defun bug.operation-vn:assign ( vn destinations )
    (loop (for operand-vn
               in (bug.depth-sort-vns (append (vn:constraining-vns vn)
                                              (vn:operand-vns      vn) ) ) )
          (bind cycle&me-list
                    (bug.vn:destinations:cycle&me-list
                        vn destinations)
                operand-destinations
                    (if (! (memq operand-vn (vn:operand-vns vn) ) ) (then
                        () )
                    (else
                        (loop (for (cycle me) in cycle&me-list) (save
                            me) ) ) ) )
    (do
        (bug.vn:assign operand-vn operand-destinations) ) )

    (let ( (cycle&me-list (bug.vn:destinations:cycle&me-list vn destinations))
           (best-cycle    () )
           (best-me       () ) )
        (if (! (cdr cycle&me-list) ) (then
            (desetq ( (best-cycle best-me) . () ) cycle&me-list) )
        (else
            (desetq (best-cycle best-me)
                (loop (for cycle&me in cycle&me-list)
                      (bind (cycle me) cycle&me)
                (minimize
                    cycle&me
                    (loop (for operand-vn in (vn:operand-vns vn) )
                    (reduce max 0
                            (me:me:delay (vn:me operand-vn) me))))))))

        (bug.vn:cycle:me:schedule vn best-cycle best-me) )
    () )


;***===========================================================================
;***
;*** (BUG.VN:DESTINATIONS:CYCLE&ME-LIST VN DESTINATIONS)
;***
;*** Given an operation VN and suggested destinations, returns a list
;*** of pairs:
;***
;***     (CYCLE ME)
;***
;*** describing the best places and times where VN can be computed; any
;*** pair in the list will calculate VN and move it to a closest
;*** destination as early as any other pair.  CYCLE specifies the time
;*** at which ME can be scheduled.
;***
;*** The result is calculated by looking at each ME that could be used
;*** for VN and calculating the cost of moving the operands to the ME,
;*** doing the operation, and then moving the value to the closest
```

```
;*** destination.
;***
;***============================================================================

(defun bug.vn:destinations:cycle&me-list ( vn destinations )
    (loop (for me in (vn:likely-mes vn) )
          (when (me:datum:ok? me vn) )
          (bind cycle (bug.vn:me:cycle vn me)
                cost  (+ cycle
                    (+ (me:delay me)
                        (bug.me:me-list:min-delay me destinations) ) ) )
    (save-minimums '(,cycle ,me) cost) ) )


;***============================================================================
;***
;*** (BUG.VN:ME:CYCLE VN ME)
;***
;*** Returns the earliest cycle in which ME can be scheduled to compute
;*** VN.  If none of VN's operands have been assigned yet, then the "cycle"
;*** returned is actually an estimate of the cost of of moving VN's
;*** operands to the ME, based on what little we know of where VN and
;*** its operands can be computed.
;***
;***============================================================================

(defun bug.vn:me:cycle ( vn me )
    (if (for-some (operand-vn in (vn:operand-vns vn) )
            (vn:bug-cycle operand-vn) )
    (then
         (bug.assigned-vn:me:cycle  vn me) )
    (else
         (bug.unassigned-vn:me:cost vn me) ) ) )


;***============================================================================
;***
;*** (BUG.UNASSIGNED-VN:ME:COST VN ME)
;***
;*** VN has none of its operands assigned.  Returns an estimate of the
;*** cost of moving the operands of VN to the ME.  The cost is computed
;*** as follows:
;***
;***     The operands of VN with maximum depth are gathered.  The cost
;***     is the maximum over those operands of F( OPERAND-VN ), where
;***     F is the delay of moving the operand to the ME.
;***
;***============================================================================

(defun bug.unassigned-vn:me:cost ( vn me )
    (let ( (max-depth (loop (for operand-vn in (vn:operand-vns vn) )
                            (reduce max 0 (vn:depth operand-vn) ) ) ) )
        (loop (for operand-vn in (vn:operand-vns vn) )
              (when (== max-depth (vn:depth operand-vn) ) )
              (reduce max 0 (bug.me-list:me:min-delay
                                  (vn:likely-mes operand-vn)
                                  me) ) ) ) ) )


;***============================================================================
;***
;*** (BUG.ASSIGNED-VN:ME:CYCLE VN ME)
;***
```

```
;*** At least one of VN's operands have been assigned.  Returns the first
;*** cycle in which VN's operation can be scheduled using ME; the cycle
;*** is the first after all the operands are available at the inputs of
;*** the ME and in which the ME is free.  (We also require that the cycle
;*** be after all the constraining VNs have completed.)
;***
;***============================================================================

(defun bug.assigned-vn:me:cycle ( vn me )
    (let ( (available-cycle
                (max
                    (loop (for operand-vn in (vn:operand-vns vn) )
                          (when (vn:bug-cycle operand-vn) )
                          (reduce max 0 (+ (bug.vn:available operand-vn)
                                           (bug.me-list:me:min-delay
                                                (vn:likely-mes operand-vn)
                                                me) ) ) )

                    (loop (for constraining-vn in (vn:constraining-vns vn) )
                          (for constraining-delay
                                               in (vn:constraining-delays vn) )
                          (when (vn:bug-cycle constraining-vn) )
                          (reduce max
                                  0
                                  (cycle:delays:constrained-cycle
                                        (vn:bug-cycle constraining-vn)
                                        (vn:delay     constraining-vn)
                                        (me:delay me)
                                        constraining-delay) ) ) ) ) )

        (cycle:resource-request:first-available-cycle
            available-cycle
            (me:resources me) ) ) )


;***============================================================================
;***
;*** (BUG.VN:AVAILABLE VN)
;***
;*** Returns the first cycle in which the value of VN is available for use.
;***
;***============================================================================

(defun bug.vn:available ( vn )
    (caseq (vn:type vn)
        ( (def copy)
            (vn:bug-cycle vn) )
        (operation
            (+ (vn:bug-cycle vn)
               (me:delay (vn:me vn) ) ) )
        (t
            (error (list vn "BUG.VN:AVAILABLE: Case error.") ) ) ) )


;***============================================================================
;***
;*** (BUG.DEPTH-SORT-VNS  VN-LIST)
;*** (BUG.HEIGHT-SORT-VNS VN-LIST)
;***
;*** Sorts a list of VNs in decreasing depth or height order.  We break
;*** ties based on the name of the VN -- this is slightly less efficient,
;*** but guarrantees greater uniformity in results when making slight
;*** changes to the input program (useful only for debugging).
```

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

(defun bug.depth-sort-vns ( vn-list )
    (sort vn-list
          (f:l ( vn1 vn2 )
            (?( (> (vn:depth vn1) (vn:depth vn2) )
                t)
              ( (== (vn:depth vn1) (vn:depth vn2) )
                (lexorder (vn:name vn1) (vn:name vn2) ) )
              ( t
                () ) ) ) ) )


(defun bug.height-sort-vns ( vn-list )
    (sort vn-list
          (f:l ( vn1 vn2 )
            (?( (> (vn:height vn1) (vn:height vn2) )
                t)
              ( (== (vn:height vn1) (vn:height vn2) )
                (lexorder (vn:name vn1) (vn:name vn2) ) )
              ( t
                () ) ) ) ) )


;***============================================================================
;***
;*** (BUG.VN:CYCLE:ME:SCHEDULE VN CYCLE ME)
;***
;*** ME has been chosen for VN.  Its resources are scheduled at CYCLE,
;*** and VN:BUG-CYCLE, VN:ME, and VN:LIKELY-MES are updated.  If ME is a
;*** register bank, we assumed it we're trying to read from it.
;***
;***============================================================================

(defun bug.vn:cycle:me:schedule ( vn cycle me )
    (assert me "BUG: No ME was found!")

    (cycle:resource-request:schedule
        cycle
        (if (== 'register-bank (me:type me) )
            (me:read-resources me)
            (me:resources      me) ) )

    (:= (vn:me          vn) me)
    (:= (vn:likely-mes vn) '(,me) )
    (:= (vn:bug-cycle  vn) cycle)
    () )


;***============================================================================
;***
;*** (BUG.ME-LIST:ME:MIN-DELAY ME)
;*** (BUG.ME:ME-LIST:MIN-DELAY ME)
;***
;*** These two functions return the minimum required delay in moving a
;*** a value between one ME and the closest ME in a list of MEs.
;***
;***============================================================================

(defun bug.me-list:me:min-delay ( me-list me )
    (if (! me-list) (then
        0)
```

```
    (else
        (loop (for next-me in me-list)
              (reduce min 100000 (me:me:delay next-me me) ) ) ) ) )


(defun bug.me:me-list:min-delay ( me me-list )
    (if (! me-list) (then
        0)
    (else
        (loop (for next-me in me-list)
              (reduce min 100000 (me:me:delay me next-me) ) ) ) ) )
```

```
;***-----------------------------------------------------------------********
;*** Copyright (C) 1983 John R. Ellis.  This file may be used for     ********
;*** educational and research purposes only, by the faculty, students, ********
;*** and staff of Yale University only.                                ********
;***-----------------------------------------------------------------********
;=========================================================================
;
; BOTTOM UP GREEDY, DEFS AND USES
;
; This file implements the part of the bottom-up-greedy algorithm that
; handles DEFs and USEs.
;
;=========================================================================

(eval-when (compile load)
    (include list-scheduler:declarations) )


;***=====================================================================
;***
;*** (BUG.DEF-VN:REASSIGN VN DESTINATION)
;***
;*** VN is a DEF VN.  One of the readers of VN has just been assigned,
;*** and this function is called to get a good guess for the location
;*** of the DEF.  DESTINATION is the ME just assigned to the reader.
;***
;*** If the DEF requires a single location and doesn't yet have one, then
;*** we find a register bank connected to DESTINATION that is available
;*** earliest and assign it to VN.  Otherwise, the DEF is allowed multiple
;*** locations and we don't assign it any location, on the assumption
;*** that we'll be able to make up "good" locations later.
;***
;*** A DEF requires a single location if it is a variable, it doesn't
;*** have any currently assigned location, and the variable is written
;*** somewhere on the trace.  Variables that aren't written on the trace
;*** are allowed to have multiple locations, as are constants.
;***
;*** *** NOTE ***
;*** We should really take account of conflicts between the register
;*** usage here, as we do later on when we assign DEFs and USEs.
;***
;***=====================================================================

(defun bug.def-vn:reassign ( vn destination )
    (if (&& (! (vn:constant?  vn) )
            (! (vn:likely-mes vn) )
            (> (dag.name:trace-write-count (vn:name vn) ) 0) )
    (then
        (assert *ls.new-defs-allowed?* "BUG: Huh?")
        (let ( (best-cycle 100000)
               (best-me    () ) )

            (loop (for me in (me:inputs destination) )
                  (when (&& (== 'register-bank (me:type me) ) )
                            (> (me:registers-left me) 0) )
                  (bind cycle (cycle:resource-request:first-available-cycle
                                   0
                                   (me:read-resources me) ) )
                  (when (< cycle best-cycle) )
            (do
                (:= best-cycle cycle)
                (:= best-me    me) ) )
```

```
            (bug.vn:cycle:me:schedule
                vn best-cycle best-me)
            (:= (me:registers-left best-me) (- &&& 1) ) ) ) ) )
    () )


;***=====================================================================
;***
;*** (BUG.MAKE-DEF1S&USES1S)
;***
;*** This procedure is called after all main recursive assignment algorithm
;*** has been run.  It inserts DEF1s between DEFs and their readers, and
;*** USE1s between USEs and their operands.
;***
;*** First, USE1s are inserted between USEs that have no given locations
;*** and their operands.  Then DEF1s are inserted between all DEFs and
;*** their readers.  Finally, USE1s are inserted between USEs that do
;*** have given locations and their operands.
;***
;***=====================================================================

(defun bug.make-def1s&use1s ()

        ;*** Make USE1s for USEs that have locations.
        ;***
    (loop (for vn in *ls.exit-vns*)
          (when (&& (== 'use (vn:type vn) )
                    (vn:locations vn) ) )
    (do
        (bug.closed-use-vn:make-use1s vn) ) )

        ;*** Make DEF1s for DEFs.
        ;***
    (loop (for vn in (bug.height-sort-vns *ls.entry-vns*) )
          (when (== 'def (vn:type vn) ) )
    (do
        (bug.def-vn:make-def1s vn) ) )


        ;*** Make USE1s for USEs that don't have locations.
        ;***
    (loop (for vn in *ls.exit-vns*)
          (when (&& (== 'use (vn:type vn) )
                    (! (vn:locations vn) ) ) )
    (do
        (bug.open-use-vn:make-use1s vn) ) )

    () )



;***=====================================================================
;***
;*** (BUG.CLOSED-USE-VN:MAKE-USE1S VN)
;***
;*** VN is a USE VN that has given locations.  A USE1 for each different
;*** location is inserted between VN and its single operand.
;***
;***=====================================================================

(defun bug.closed-use-vn:make-use1s ( vn )
    (let ( (use1-vns
                (loop (for (me register) in (vn:locations vn) ) (save
```

```
                    (vn:create (vn:new
                       type             'use1
                       name             (vn:name      vn)
                       constant?        (vn:constant? vn)
                       datatype         (vn:datatype  vn)
                       bug-cycle        (vn:bug-cycle vn)
                       height           (vn:height    vn)
                       depth            (vn:depth     vn)
                       likely-mes       '(,me)
                       me               me
                       register-bank-me me
                       register         register) ) ) ) ) )

        (vn:insert-vns vn use1-vns)
        () ) )


;***========================================================================
;***
;*** (BUG.OPEN-USE-VN:MAKE-USE1S VN)
;***
;*** VN is a USE VN that has no given locations.  If the USE doesn't have
;*** a DEF1 as its operand, that means the variable of the USE is written
;*** on the trace; we splice a USE1 betweeen the operand and the USE.
;***
;*** If the USE does have a DEF1 as its operand, that means the variable
;*** (or constant) isn't written on the trace.  We splice a USE1 between
;*** ALL the DEF1s of the variable.  The effect of this is to keep all
;*** the copies of this variable live over the whole trace (good if the
;*** trace is a loop -- the variable or constant is a loop invariant).
;*** Perhaps we'll want to modify this to do it only when given a loop
;*** trace.
;***
;***========================================================================

(defun bug.open-use-vn:make-use1s ( vn )
    (let*( (operand-vn (car (vn:operand-vns vn) ) )
           (pred-vns
               (if (== 'def1 (vn:type operand-vn) ) (then
                   (vn:reading-vns (car (vn:operand-vns operand-vn) ) ) )
               (else
                   (vn:operand-vns vn) ) ) ) )

        (loop (for pred-vn in pred-vns)
              (bind use1-vn (vn:create (vn:new
                                type             'use1
                                name             (vn:name      vn)
                                constant?        (vn:constant? vn)
                                datatype         (vn:datatype  vn)
                                bug-cycle        (vn:bug-cycle vn)
                                height           (vn:height    vn)
                                depth            (vn:depth     vn) ) ) )
        (do
            (if (memq pred-vn (vn:operand-vns vn) ) (then
                (vn:splice-vn pred-vn use1-vn (list vn) ) )
            (else
                (push (vn:reading-vns pred-vn) use1-vn)
                (push (vn:operand-vns use1-vn) pred-vn)
                (push (vn:reading-vns use1-vn) vn)
                (push (vn:operand-vns vn)     use1-vn) ) ) ) )
        () ) )
```

```
;***========================================================================
;***
;*** (BUG.DEF-VN:MAKE-DEF1S VN)
;***
;*** VN is a DEF.  DEF1s are inserted between the DEF and all its readers,
;*** picking likely locations (register banks or constant generators)
;*** and assigning them to the DEF1s.
;***
;*** Each reader of the DEF is considered in turn, and a good location
;*** for that reader is chosen.  Then all the readers with the same
;*** location are grouped togethor, and a single DEF1 for that location
;*** is spliced between the DEF and the group of readers.
;***
;***========================================================================

(defun bug.def-vn:make-def1s ( vn )
    (let ( (me-table () )
           (all-reading-vns
                (append (bug.height-sort-vns (vn:reading-vns vn) )
                        (vn:off-live-reading-vns vn) ) )
           (number-reading-vns
                (length (vn:reading-vns vn) ) ) )

            ;*** Pick good locations for the DEF by picking a good location
            ;*** that each of the readers would prefer.  Keep a table
            ;*** that groups all the readers of a preferred ME together.
            ;*** As locations are picked, keep track of the number of
            ;*** registers left in register banks.  We have to treat
            ;*** off-live readers almost like normal readers, because
            ;*** they need locations for the DEF as well.
            ;***
        (loop (for reading-vn in all-reading-vns)
             (incr i from 1)
              (bind reading-vn? (<= i number-reading-vns)
                    me          (bug.def-vn:reading-vn:choose-me
                                    vn
                                    (if reading-vn? reading-vn () ) ) )
        (do
            (if (! (memq me (vn:likely-mes vn) ) ) (then
                (push (vn:likely-mes vn) me)
                (caseq (me:type me)
                    (register-bank
                        (:= (me:registers-left me) (- &&& 1) ) )
                    (constant-generator
                        (cycle:resource-request:schedule
                            (cycle:resource-request:first-available-cycle
                                0 (me:resources me) )
                            (me:resources me) ) )
                    (t
                        (error (list me "Case error.") ) ) ) ) )

            (let ( (me-entry (assoc me me-table) ) )
                (if (! me-entry) (then
                    (:= me-entry '(,me () () ) )
                    (push me-table me-entry) ) )

                (if reading-vn? (then
                    (push (cadr  me-entry) reading-vn) )
                (else
                    (push (caddr me-entry) reading-vn) ) ) ) ) )

            ;*** Now for each assigned location of the DEF, we have
            ;*** a group of readers.   Insert a DEF1 between the DEF and
```

```
            ;*** readers.
            ;***
        (loop (for (me reading-vns off-live-reading-vns) in me-table)
              (bind (() register) (assoc me (vn:locations vn) ) )
              (bind defi-vn (vn:create (vn:new
                                  type        'defi
                                  name        (vn:name      vn)
                                  constant?   (vn:constant? vn)
                                  datatype    (vn:datatype  vn)
                                  bug-cycle   (vn:bug-cycle vn)
                                  height      (vn:height    vn)
                                  depth       (vn:depth     vn)
                                  likely-mes  '(,me)
                                  me          me) ) )
        (do
            (if register (then
                (:= (vn:register-bank-me defi-vn) me)
                (:= (vn:register         defi-vn) register) ) )
            (vn:splice-vn vn defi-vn reading-vns off-live-reading-vns) ) )
        () ) )


;***=========================================================================
;***
;*** (BUG.DEF-VN:READING-VN:CHOOSE-ME VN READING-VN)
;***
;*** Chooses a good location for VN, a DEF, to be read by one its children,
;*** READING-VN.  The choice is made depending on whether or not the DEF
;*** is a constant or variable, and whether or not we are allowed to make
;*** up new locations for the DEF or must stick with the locations given.
;***
;***=========================================================================

(defun bug.def-vn:reading-vn:choose-me ( vn reading-vn )
    (if (vn:constant? vn) (then
        (if *ls.new-defs-allowed?* (then
            (bug.def-open-constant-vn:reading-vn:choose-me   vn reading-vn) )
        (else
            (bug.def-closed-constant-vn:reading-vn:choose-me vn reading-vn))))
    (else
        (if (&& *ls.new-defs-allowed?*
                (== (dag.name:trace-write-count (vn:name vn) ) 0) )
        (then
            (bug.def-open-var-vn:reading-vn:choose-me   vn reading-vn) )
        (else
            (bug.def-closed-var-vn:reading-vn:choose-me vn reading-vn))))))


;***=========================================================================
;***
;*** (BUG.DEF-OPEN-CONSTANT-VN:READING-VN:CHOOSE-ME VN READING-VN)
;***
;*** VN is a constant DEF for which we are allowed to make up new
;*** locations.  If the constant is immediate and we're supposed to load
;*** immediates from constant generators, then we look for the closest
;*** such generator.  Otherwise, we treat the DEF just as if it were a
;*** register, assigning it a register location.
;***
;***=========================================================================

(defun bug.def-open-constant-vn:reading-vn:choose-me ( vn reading-vn )
    (let ( (mes (constant:constant-generator-mes (vn:name vn) ) ) )
        (if (&& mes
```

```
                (== 'load *ls.immediate-constant-action*) )
        (then
            (car (bug.def-constant-vn:reading-vn:best-cg-me&cycle
                    vn reading-vn) ) )
        (else
            (bug.def-open-var-vn:reading-vn:choose-me vn reading-vn) ) ) ) )


;***=========================================================================
;***
;*** (BUG.DEF-CLOSED-CONSTANT-VN:READING-VN:CHOOSE-ME VN READING-VN)
;***
;*** VN is a constant DEF for which we are not allowed to make up new locations.
;*** We consider two options:
;***
;***      1.  Using an already assigned location.
;***      2.  Using a constant generator.
;***
;*** We pick the one that is closest to READING-VN, preferring #1 over #2 if
;*** there is a choice (i.e. preferring a register bank over a constant
;*** generator, hmmm).
;***
;***=========================================================================

(defun bug.def-closed-constant-vn:reading-vn:choose-me ( vn reading-vn )
    (let*( (dest-me    (bug.vn:dest-me reading-vn) )
           (best-delay 100000)
           (best-me    () ) )

        (loop (for (me register) in (vn:locations vn) )
              (bind delay (me:me:delay me dest-me) )
              (when (|| (== me dest-me)
                        (< delay best-delay) ) )
        (do
            (:= best-delay delay)
            (:= best-me    me) ) )

        (let ( ( (best-cg-me best-cg-delay)
                   (bug.def-constant-vn:reading-vn:best-cg-me&cycle
                       vn reading-vn) ) )
            (if (&& best-cg-me
                    (< best-cg-delay best-delay) )
            (then
                (:= best-me best-cg-me) ) ) )

        (assert best-me
            (h vn) t (h reading-vn)
            "BUG.DEF-CLOSED-CONSTANT-VN:READING-VN:CHOOSE-ME: Wasn't able "
            "to find a register or constant generator for VN!")

        best-me) )


(defun bug.def-constant-vn:reading-vn:best-cg-me&cycle
        ( vn reading-vn )

    (let ( (dest-me (bug.vn:dest-me reading-vn) ) )

        (loop (for me in (constant:constant-generator-mes (vn:name vn) ) )
              (bind cycle (+ (cycle:resource-request:first-available-cycle
                                   0
                                   (me:resources me) )
```

```
                             (me:me:delay me dest-me) ) )
        (minimize '(,me ,cycle) cycle) ) ) )


;***============================================================================
;***
;*** (BUG.DEF-OPEN-VAR-VN:READING-VN:CHOOSE-ME   VN READING-VN)
;*** (BUG.DEF-CLOSED-VAR-VN:READING-VN:CHOOSE-ME VN READING-VN)
;***
;*** VN is a DEF of a variable.  These two functions pick a good register
;*** bank location for it, the former considering any register bank, the
;*** latter considering only register banks already assigned to the VN.
;***
;***============================================================================

(defun bug.def-open-var-vn:reading-vn:choose-me ( vn reading-vn )
    (bug.def-var-vn:reading-vn:choose-me
        vn
        reading-vn
        *ls.register-bank-mes*) )


(defun bug.def-closed-var-vn:reading-vn:choose-me ( vn reading-vn )
    (bug.def-var-vn:reading-vn:choose-me
        vn
        reading-vn
        (if reading-vn
            (vn:likely-mes vn)
            *ls.register-bank-mes*) ) )
                ;***
                ;*** It's possible to have a closed VN (a variable written
                ;*** on the trace) that isn't assigned any likely ME at
                ;*** this point.  This only occurs if READING-VN is (),
                ;*** that is, if VN doesn't have any readers, just off-live
                ;*** readers.

;***============================================================================
;***
;*** (BUG.DEF-VAR-VN:READING-VN:CHOOSE-ME VN READING-VN FEASIBLE-MES)
;***
;*** VN is a DEF of a variable.  Chooses a good register bank location
;*** for it from FEASIBLE-MES, a list of register banks.  We first find
;*** the register banks capable of holding VN that are closest to
;*** READING-VN.  Then of those, we find the ones that conflict least
;*** with the other operands of READING-VN that already have locations.
;*** Then if there is still a choice left, we prefer a register bank that
;*** is either identical with the destination of READING-VN (if it is
;*** a USE1) or one that is already assigned to VN.
;***
;***============================================================================

(defun bug.def-var-vn:reading-vn:choose-me ( vn reading-vn feasible-mes )
    (let*( (dest-me
                (bug.vn:dest-me reading-vn) )

           (close-mes
                (loop (for me in feasible-mes)
                      (when (|| (memq me (vn:likely-mes vn) )
                                (> (me:registers-left me) 0) ) )
                      (bind delay (me:me:delay me dest-me) )
                (save-minimums me delay) ) )

           (least-conflicting-mes
```

```
                (if (! reading-vn) (then
                    close-mes)
                (else
                    (loop (for me in close-mes) (save-minimums
                        me
                        (bug.vn:me:register-conflicts reading-vn me) ) ) ) ) )

           (best-me
                (if (memq dest-me least-conflicting-mes) (then
                    dest-me)
                (else
                    (if-let ( (common-mes (intersectionq least-conflicting-mes
                                                         (vn:likely-mes vn))))
                        (car common-mes)
                        (car least-conflicting-mes) ) ) ) ) )

        (assert best-me
            (h vn) t (h reading-vn) t
            "BUG.DEF-VAR-VN:READING-VN:CHOOSE-ME: Wasn't able to "
            " find a close register bank!")

        best-me) )


;***============================================================================
;***
;*** (BUG.VN:DEST-ME VN)
;***
;*** Returns the destination ME that an operand of VN needs to deliver its
;*** result to.  The ME depends on the type of VN:
;***
;***     OPERATION:    the functional unit assiged to the VN.
;***     COPY:         wherever the max height reader of VN wants its value.
;***     USE1:         any register bank assigned to the use.
;***
;*** Returning () means that we don't have any particular preference for a
;*** a destination ME; this should only occur if VN is a USE1 and doesn't have
;*** a required location.    VN may be (), in which case the result is
;*** ().
;***
;***============================================================================

(defun bug.vn:dest-me ( vn )
    (&& vn
        (caseq (vn:type vn)
            (operation
                (vn:me vn) )
            (copy
                (bug.vn:dest-me (loop (for reading-vn in (vn:reading-vns vn) )
                                       (maximize reading-vn
                                                   (vn:height reading-vn)))))
            (use1
                (car (vn:likely-mes vn) ) )
            (use
                (car (vn:likely-mes vn) ) )
            (t
                (error (list vn "BUG.VN:DEST-ME") ) ) ) ) )


;***============================================================================
;***
;*** (BUG.VN:ME:REGISTER-CONFLICTS VN ME)
;***
```

```
;*** Attempts to determine whether by using ME (a register bank) as a
;*** location for one of the as-yet unassigned operands of VN, that will
;*** conflict with the already assigned operands.  In general, an
;*** assignment of register banks to the operands of VN conflicts if it
;*** is not possible to read all the operands from the register banks
;*** in the same cycle (e.g. because there are not enough register ports).
;***
;*** Currently, we assume that the only possible resource that could cause
;*** conflict is the number of available ports (i.e. all ports of of a
;*** bank are connected to the identical set of destinations).  So checking
;*** to see if ME conflicts with previously assigned operands is merely
;*** a matter of counting the number of ME ports required by all the
;*** operands.
;***
;*** Eventually, we might also consider conflict of ME:ME:RESOURCES as
;*** well.
;***
;*** Returns 1 if there is conflict, 0 if not (so we can add them up?).
;***
;***=============================================================================

(defun bug.vn:me:register-conflicts ( vn me )
    (let ( (required-ports
                 (loop (for operand-vn in (vn:operand-vns vn) )
                       (when (&& (!== 'def (vn:type operand-vn) )
                                 (== me (car (vn:likely-mes vn) ) ) ) )
                       (reduce + 1 1) ) ) )
        (if (> required-ports (me:read-ports me) )
            1
            0) ) )
```

```
;***-------------------------------------------------------------------********
;*** Copyright (C) 1983 John R. Ellis.  This file may be used for      ********
;*** educational and research purposes only, by the faculty, students, ********
;*** and staff of Yale University only.                                ********
;***-------------------------------------------------------------------********

(:= *ls.build-module-list* '(

    list-scheduler:resource
    list-scheduler:machine-model
    list-scheduler:shortest-path

    list-scheduler:vn
    list-scheduler:dag
    list-scheduler:heights-depths

    list-scheduler:bug
    list-scheduler:bug-defs-uses

    list-scheduler:scheduler
    list-scheduler:sch-operation
    list-scheduler:sch-copy
    list-scheduler:sch-splits-joins
    list-scheduler:registers

    list-scheduler:mis-to-eli
    list-scheduler:simulator
    list-scheduler:lsex-options
    list-scheduler:lsex
    ) )

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

```
;***-------------------------------------------------------------------********
;*** Copyright (C) 1983 John R. Ellis.  This file may be used for     ********
;*** educational and research purposes only, by the faculty, students, ********
;*** and staff of Yale University only.                                ********
;***-------------------------------------------------------------------********
;*** Build the latest version of the list-scheduling compiler.
;***

(remprop 'loop 'build-information)      ;*** Get rid of Yale Loop info.

(:= *build-module-list* () )

(load 'utilities:build)                 ;*** This must go first.
(load 'interpreter:build)
(load 'trace:build)
(load 'diophantine:build)
(load 'flow-analysis:build)
(load 'experiments:build)
(load 'list-scheduler:build)

(build)

(load 'list-scheduler:eli-model)
```

```
;***--------------------------------------------------------------********
;*** Copyright (C) 1988 John R. Ellis.  This file may be used for  ********
;*** educational and research purposes only, by the faculty, students, ********
;*** and staff of Yale University only.                             ********
;***--------------------------------------------------------------********
;=========================================================================
;
; DAG Making
;
; This module constructs a dag of VNs from the trace.
;
;
; (DAG.INITIALIZE)
;     Initializes this module.
;
; (DAG.MAKE LIVE-BEFORE SOURCE-RECORD-LIST LIVE-AFTER)
;     Constructs a dag of VNs from this information (parameters identical
;     in meaning to those of GENERATE-CODE).
;
; *LS.ENTRY-VNS*
; *LS.EXIT-VNS*
;     Lists of all VNs with no predecessors and no successors,
;     respectively, of any kind.
;
; *LS.NEW-DEFS-ALLOWED?*
;     True if on this trace we are allowed to make up new locations for
;     DEFs (that is, if we weren't given any DEF at the beginning of the
;     trace or if this trace is not the beginning trace of a program.)
;
; *LS.LOOP-TRACE?*
;     True if this trace begins with a LOOP-START and ends with a
;     TRACE-FENCE pseudo-op, indicating the trace is a body of a loop
;     (hack hack).
;
; (DAG.NAME:TRACE-WRITE-COUNT NAME)
;     Returns the number of times NAME was written by an operation on the
;     trace.
;
; (DAG.NAME:USE-VN NAME)
;     Returns a USE VN of the given NAME if such a VN exists, () otherwise.
;
; (DAG.SET-COUNTS)
;     Sets the predecessors-left and readers-left counts of each VN.
;
; (CYCLE:DELAYS:CONSTRAINED-CYCLE CYCLE PD SD CD)
;     Returns the cycle that a constrained successor can earliest be
;     scheduled, given the CYCLE of its constraining predecessor, the
;     predecessor's delay PD, the successor's delay SD, and the constraining
;     delay betwen them CD.  The result cycle is defined as:
;
;         MAX( CYCLE + PD - SD + CD, CYCLE + CD )
;
;     This guarrantees that the first cycle of the successor must be at
;     least CD cycles after the first cycle of the predecessor, and the
;     last cycle of the successor at least CD cycles after the last cycle
;     of the predecessor.  These requirements arise from the nature of
;     partial schedules -- a write of a variable must not allowed to be
;     copied up into a rejoin above a predecessor read of the same variable
;     that is in the rejoion partial schedule (an analagous situation holds
;     for splits).
;
; (CYCLE:DELAYS:CONSTRAINING-CYCLE CYCLE PD SD CD)
;     Returns the cycle that a constraining predecessor can latest be
```

```
;     scheduled given the CYCLE of its constrained successor, the delay
;     of the predecessor, the delay of the successor, and the constraining
;     delay between them.
;
;=========================================================================

(eval-when (compile load)
    (include list-scheduler:declarations) )

(declare (special
    *dag.name:vn*          ;*** Hash table mapping names to the most recent
                           ;*** VN containing that name.  Uses EQUALT so
                           ;*** that numbers are considered equal only if they
                           ;*** are of the same type.
    *dag.name:write-count*
                           ;*** Hash table mapping names to the number of times
                           ;*** they were written on the trace.
    *dag.name:locations*
                           ;*** Hash table mapping names to locations given in
                           ;*** a DEF pseudo-op at the beginning of the trace.

    *dag.name:use-vn*      ;*** Hash table mapping names to USE VNs.

    *dag.live-before*      ;*** The live-before variables.
    *dag.live-after-constants*
                           ;*** Constants read on the trace which we want to
                           ;*** treat as live-after.

    *dag.loop-start?*      ;*** True if this trace contains a LOOP-START?

    *dag.use?*             ;*** True if we saw a non-empty USE.
    ) )


(defun dag.initialize ()
    (:= *ls.entry-vns*              () )
    (:= *ls.exit-vns*               () )
    (:= *ls.new-defs-allowed?*      t)
    (:= *ls.loop-trace?*            () )
    (:= *dag.name:vn*               () )
    (:= *dag.name:write-count*      () )
    (:= *dag.name:locations*        () )
    (:= *dag.live-before*           () )
    (:= *dag.live-after-constants*  () )
    (:= *dag.loop-start?*           () )
    (:= *dag.use?*                  () )
    () )


(defun dag.make ( live-before source-record-list live-after )
    (dag.initialize)
    (start-trace)

    (:= *dag.name:vn*          (hash-table:create 'equalt () () ) )
    (:= *dag.name:use-vn*      (hash-table:create 'equalt () () ) )
    (:= *dag.name:write-count* (hash-table:create ()      () 0) )
    (:= *dag.name:locations*   (hash-table:create 'equalt () () ) )
    (:= *dag.live-before*      live-before)

        ;*** Make VNs for the trace operations.
        ;***
    (loop (for source-record in source-record-list) (do
        (dag.source-record:create-vn source-record) ) )
```

```
        ;*** Make VNs for the live after variables and constants, including
        ;*** the constants read on the trace if there isn't a non-empty
        ;*** USE at the end.
        ;***
    (if (! *dag.use?*) (then
        (:= live-after (unionq &&& *dag.live-after-constants*) ) ) )
    (loop (for name in live-after)
          (when (!== '%trace name) )
    (do
        (dag.live-name:create-use name) ) )

        ;*** Set the successor fields of the VNs.
        ;***
    (dag.set-successor-vns)

        ;*** Delete dead and useless VNs.
        ;***
    (dag.remove-dead-vns)

        ;*** Gather the lists of entry and exit VNs.
        ;***
    (loop (for-each-vn vn) (do
        (if (&& (! (vn:operand-vns      vn) )
                (! (vn:constraining-vns vn) ) )
        (then
            (push *ls.entry-vns* vn) ) )

        (if (&& (! (vn:reading-vns     vn) )
                (! (vn:constrained-vns vn) ) )
        (then
            (push *ls.exit-vns* vn) ) ) ) )

    () )


;***=========================================================================
;***
;*** (DAG.SOURCE-RECORD:CREATE-VN
;***      (OPER TRACE-DIRECTION DATUM OFF-LIVE) )
;***
;*** Processes one source-record from the trace, creating an OPER, DEF,
;*** or, USE VN for that record.
;***
;***=========================================================================

(defun dag.source-record:create-vn ( source-record )
    (let*( ( (oper trace-direction datum off-live)
                source-record)
           ( operator (oper:operator oper) )
           ( group    (oper:group    oper) ) )

        (?      ;*** A DEF?
                ;***
          ( (== 'def operator)
            (loop (for (name me register) in (oper:part oper 'body) )
                  (when (!== '%trace name) )
            (do
                (:= *ls.new-defs-allowed?* () )
                (push ([]h *dag.name:locations* name)
                    '(,(name:me me) ,register) ) ) ) )

                ;*** A USE?
```

```
                ;***
          ( (== 'use operator)
            (loop (for use-spec in (oper:part oper 'body) )
                  (when (!== '%trace (car use-spec) ) )
            (do
                (:= *dag.use?* t)
                (dag.use-spec:create-use use-spec) ) ) )

                ;*** A DEF-BLOCK?
                ;***
          ( (== 'def-block operator)
            (:= *ls.new-defs-allowed?* () )
            (dag.source-record:create-pseudo-op source-record) )

                ;*** A LOOP-START?
                ;***
          ( (== 'loop-start operator)
            (:= *dag.loop-start?* t)
            (predecessors oper trace-direction () ) )

                ;*** A TRACE-FENCE?
                ;***
          ( (== 'trace-fence operator)
            (:= *ls.loop-trace?* *dag.loop-start?*)
            (predecessors oper trace-direction () ) )

                ;*** An IN-PARAMETER?
                ;***
          ( (== 'in-parameter group)
            (dag.source-record:create-in-parameter source-record) )

                ;*** An OUT-PARAMETER?
                ;***
          ( (== 'out-parameter group)
            (dag.source-record:create-out-parameter source-record) )

                ;*** Some random pseudo op?
                ;***
          ( (oper:property? oper 'pseudo-op)
            (if (memq operator '(def-block dcl esc assert) )
            (then
                (dag.source-record:create-pseudo-op source-record) )
            (else
                (predecessors oper trace-direction () ) ) ) )

                ;*** An "real" normal operation.
                ;***
          ( t
            (dag.source-record:create-operation source-record) ) )

        () ) )


;***=========================================================================
;***
;*** (DAG.SOURCE-RECORD:CREATE-PSEUDO-OP
;***      (OPER TRACE-DIRECTION DATUM OFF-LIVE) )
;***
;*** Processes one pseudo-operation source-record from the trace, making
;*** a PSEUDO-OP VN for that record.
;***
;***=========================================================================
```

```
(defun dag.source-record:create-pseudo-op
       ( (oper trace-direction datum off-live) )

    (let ( (vn (vn:create (vn:new
                    type            'pseudo-op
                    oper            oper
                    datum           datum) ) ) )

         (predecessors oper trace-direction vn)
         vn) )


;***=====================================================================
;***
;*** (DAG.SOURCE-RECORD:CREATE-IN-PARAMETER
;***     (OPER TRACE-DIRECTION DATUM OFF-LIVE) )
;***
;*** Processes an xIN operation representing an input parameter by generating
;*** an FPLOAD that reads in the operand.
;***
;***=====================================================================

(defun dag.source-record:create-in-parameter
       ( (oper trace-direction datum off-live) )

    (if (> (name:rank (oper:part oper 'written) ) 0) (then
        () )
    (else
        (let*( (datatype   (oper:dest-datatype oper) )
               (written    (oper:part oper 'written) )
               (address-vn (dag.operand:vn `(address ,written) ) )
               (load-op    (caseq datatype
                                (float   'fpload)
                                (integer 'ipload) ) )
               (vn         (vn:create (vn:new
                                type            'operation
                                name            written
                                oper            `(,load-op () () )
                                operand-vns     `(,address-vn)
                                datum           datum) ) ) )

            (:= ([]h *dag.name:vn* written) vn)
            vn) ) ) )


;***=====================================================================
;***
;*** (DAG.SOURCE-RECORD:CREATE-OUT-PARAMETER
;***     (OPER TRACE-DIRECTION DATUM OFF-LIVE) )
;***
;*** Processes an xOUT operation representing an output parameter by
;*** generating an FPSTORE that stores the operand.
;***
;***=====================================================================

(defun dag.source-record:create-out-parameter
       ( (oper trace-direction datum off-live) )

    (if (> (name:rank (oper:part oper 'read1) ) 0) (then
        () )
    (else
        (let*( (datatype   (oper:dest-datatype oper) )
               (read1      (oper:part oper 'read1) )
```

```
               (address-vn (dag.operand:vn `(address ,read1) ) )
               (read-vn    (dag.operand:vn read1) )
               (store-op   (caseq datatype
                                (float   'fpstore)
                                (integer 'ipstore) ) )
               (vn         (vn:create (vn:new
                                type            'operation
                                oper            `(,store-op () () )
                                operand-vns     `(,address-vn ,read-vn)
                                datum           datum) ) ) )
            (assert read-vn "DAG: Operand doesn't have a VN.")
            vn) ) ) )


;***=====================================================================
;***
;*** (DAG.SOURCE-RECORD:CREATE-OPERATION
;***     (OPER TRACE-DIRECTION DATUM OFF-LIVE) )
;***
;*** Processes one operation source-record from the trace, making an
;*** OPERATION VN for that record.
;***
;***=====================================================================

(defun dag.source-record:create-operation
       ( (oper trace-direction datum off-live) )

    (let*( (vn (vn:create (vn:new
                    type    (if (memq (oper:operator oper)
                                      '(iassign fassign vbase iconstant
                                        fconstant) )
                                'copy
                                'operation)
                    datum   datum
                    name    (if (== 'vstore (oper:group oper) )
                                ()
                                (oper:part oper 'written) )
                    oper    oper) ) )
           (pred-list (predecessors oper trace-direction vn) ) )

            ;*** Construct the operands read by this operation.
            ;***
        (:= (vn:operand-vns vn) (dag.oper:operand-vns oper) )

            ;*** Construct the scheduling constraints.
            ;***
        (let ( ( (vns delays)
                    (dag.pred-list:constraining-vns&delays pred-list) ) )
            (:= (vn:constraining-vns vn)     vns)
            (:= (vn:constraining-delays vn) delays) )

            ;*** Construct the list of off-live vns.
            ;***
        (:= (vn:off-live-vns vn)
            (loop (for name in off-live)
                  (when (!== '%trace name) )
                  (bind off-live-vn (dag.operand:vn name) )
            (save off-live-vn) ) )

            ;*** If this VN writes a value, then record the VN under
            ;*** the variable name written.
            ;***
        (if (vn:name vn) (then
```

```
            (:= ([]h *dag.name:vn* (vn:name vn) ) vn)
            (:= ([]h *dag.name:write-count* (vn:name vn) )
                (+ 1 &&&) ) ) )
        () ) )


;***=======================================================================
;***
;*** (DAG.OPER:OPERAND-VNS OPER)
;***
;*** Returns the list of operand VNs read by OPER.
;***
;***=======================================================================

(defun dag.oper:operand-vns ( oper )
    (caseq (oper:operator oper)
        ( (ipload fpload)
            '(,(dag.operand:vn (oper:part oper 'index) ) ) )

        ( (ipstore fpstore)
            '(,(dag.operand:vn (oper:part oper 'index) )
              ,(dag.operand:vn (oper:part oper 'value) ) ) )

        ( (ivload fvload)
            '(,(dag.operand:vn '(vbase ,(oper:part oper 'vector) ) )
              ,(dag.operand:vn (oper:part oper 'index) ) ) )

        ( (ivstore fvstore)
            '(,(dag.operand:vn '(vbase ,(oper:part oper 'vector) ) )
              ,(dag.operand:vn (oper:part oper 'index) )
              ,(dag.operand:vn (oper:part oper 'value) ) ) )

        (vbase
            '(,(dag.operand:vn '(vbase ,(oper:part oper 'vector) ) ) ) )

        ( t
            (loop (for operand in (oper:part oper 'read) ) (save
                (dag.operand:vn operand) ) ) ) ) )


;***=======================================================================
;***
;*** (DAG.OPERAND:VN OPERAND)
;***
;*** Returns the VN corresponding to OPERAND, which is either a variable
;*** or a constant.  If a previous VN for the variable or constant exists,
;*** it is returned.   Otherwise, one is created.
;***
;***=======================================================================

(defun dag.operand:vn ( operand )
    (if-let ( (vn ([]h *dag.name:vn* operand) ) ) (then
        vn)
    (else
        (if (litatom operand) (then
            (assert (memq operand *dag.live-before*)
                (h operand) " DAG: Name not mentioned in LIVE-BEFORE list.")
            (:= ([]h *dag.name:vn* operand)
                (vn:create (vn:new
                    type      'def
                    name      operand
                    locations ([]h *dag.name:locations* operand) ) ) ) )
        (else
```

```
            (dag.constant:create-vn operand) ) ) ) ) )


;***=======================================================================
;***
;*** (DAG.CONSTANT:CREATE-VN CONSTANT)
;***
;*** Creates a new VN for a constant (we know there isn't currently one).
;***
;***=======================================================================

(defun dag.constant:create-vn ( constant )
    (let ( (datatype   (caseq (typep constant)
                            ( (double flonum) 'float)
                            ( t               'integer) ) )
           (locations  ([]h *dag.name:locations* constant) )
           (immediate? (constant:constant-generator-mes constant) ) )

            ;*** If desired, add this constant to the live-after list of
            ;*** variables to cause it to be retained for the whole trace.
            ;***
        (if (if immediate?
                (== 'retained-register *ls.immediate-constant-action*)
                (== 'retained-register *ls.memory-constant-action*) )
        (then
            (if (for-every (x in *dag.live-after-constants*)
                    (! (equalt x constant) ) )
            (then
                (push *dag.live-after-constants* constant) ) ) ) )

        (?      ;*** Any locations for the constant given in the DEF at
                ;*** the beginning of the trace?  If so, use them.
                ;***
          ( locations
            (:= ([]h *dag.name:vn* constant)
                (vn:create (vn:new
                    type       'def
                    constant?  t
                    name       constant
                    locations  locations) ) ) )

                ;*** No constant generator can make this constant and
                ;*** we're not allowed to make up new DEF locations?  Make
                ;*** an xPLOAD of the constant from memory.
                ;***
          ( (&& (|| (! *ls.new-defs-allowed?*)
                    (== 'load *ls.memory-constant-action*) )
                (! immediate?) )
            (let ( (address-vn (dag.operand:vn '(address ,constant) ) )
                   (load-op    (caseq datatype
                                    (float   'fpload)
                                    (integer 'ipload) ) ) )
                (:= ([]h *dag.name:vn* constant)
                    (vn:create (vn:new
                        type        'operation
                        name        constant
                        oper        '(,load-op () () )
                        operand-vns '(,address-vn) ) ) ) ) )

                ;*** Just make a normal constant VN.
                ;***
          ( t
            (:= ([]h *dag.name:vn* constant)
```

```
                (vn:create (vn:new
                    type      'def
                    constant? t
                    name      constant) ) ) ) ) ) )


;***=======================================================================
;***
;*** (DAG.LIVE-NAME:CREATE-USE NAME)
;***
;*** Makes a USE VN for a name that is live on exit from the trace.  A
;*** USE VN may already exist for NAME, created by a USE on the trace.
;***
;***=======================================================================

(defun dag.live-name:create-use ( name )
    (if-let ( (vn ([]h *dag.name:use-vn* name) ) ) (then
        vn)
    (else
        (:= ([]h *dag.name:use-vn* name)
            (vn:create (vn:new
                        type        'use
                        name        name
                        operand-vns '(,(dag.operand:vn name) ) ) ) ) ) ) )


;***=======================================================================
;***
;*** (DAG.USE-SPEC:CREATE-USE (NAME ME REGISTER) )
;***
;*** The list (NAME ME REGISTER) occurred in a USE in the trace.  A USE
;*** VN for NAME is created if one doesn't exist for it already, and it
;*** is updated by the specified value location.
;***
;***=======================================================================

(defun dag.use-spec:create-use ( (name me register) )
    (if-let ( (vn ([]h *dag.name:use-vn* name) ) ) (then
        (push (vn:locations vn) '(,(name:me me) ,register) )
        vn)
    (else
        (:= ([]h *dag.name:use-vn* name)
            (vn:create (vn:new
                 type            'use
                 name            name
                 operand-vns     '(,(dag.operand:vn name) )
                 locations       '( (,(name:me me) ,register) ) ) ) ) ) ) )


;***=======================================================================
;***
;*** (DAG.PRED-LIST:CONSTRAINING-VNS&DELAYS PRED-LIST)
;***
;*** PRED-LIST is the result returned by a call to PREDECESSORS that
;*** describes the predecessors of an operation.   This returns the
;*** VNs that constrain the scheduling of an operation (e.g. conditional-
;*** conflict VNs or write-after-write VNs);  operand VNs are not included.
;***
;*** The result has the form:
;***
;***    (VNS DELAYS)
;***
;*** where VNS is a list of predecessor VNs and DELAYS is a corresponding
```

```
;*** list of numbers saying how much later the current VN must be done
;*** after the predecessor VN.
;***
;***=======================================================================

(defun dag.pred-list:constraining-vns&delays ( pred-list )
    (loop (for (pred reason source-operand source-type pred-operand pred-type)
               in pred-list)
          (initial vns    ()
                   delays () )
    (do
        (caseq reason
            ( (conditional-conflict possible-operand-conflict)
                (push vns   pred)
                (push delays 1) )

            (operand-conflict
                (?( (&& (== 'written source-type)
                        (== 'read    pred-type) )
                    (push vns   pred)
                    (push delays 0) )

                  ( (&& (== 'written source-type)
                        (== 'written pred-type) )
                    (push vns   pred)
                    (push delays 1) )

                  ( (&& (== 'read    source-type)
                        (== 'written pred-type) )
                    (let ( (pred-oper (vn:oper pred) ) )
                        (if (&& (oper:property? pred-oper 'vector-reference)
                               (== (oper:part pred-oper 'vector)
                                   (nth-elt (cdr pred-oper) pred-operand)))
                        (then
                            (push vns   pred)
                            (push delays 1) ) ) ) ) ) )

            (t
                (error (list pred-list "DAG: Invalid pred-list.") ) ) ) )
    (result
        '(,(dreverse vns)
          ,(dreverse delays) ) ) ) )


;***=======================================================================
;***
;*** (DAG.SET-SUCCESSOR-VNS)
;***
;*** Sets the successor fields of each VN from its successors' predecessor
;*** fields:
;***
;***     :READING-VNS          from :OPERAND-VNS
;***     :CONSTRAINED-VNS      from :CONSTRAINING-VNS
;***     :OFF-LIVE-READING-VNS from :OFF-LIVE-VNS
;***
;***=======================================================================

(defun dag.set-successor-vns ()

    (loop (for-each-vn vn) (do
        (loop (for operand-vn in (vn:operand-vns vn) ) (do
            (push (vn:reading-vns operand-vn) vn) ) )
```

```
        (loop (for off-live-vn in (vn:off-live-vns vn) ) (do
            (push (vn:off-live-reading-vns off-live-vn) vn) ) )
        (loop (for constraining-vn in (vn:constraining-vns vn) ) (do
            (push (vn:constrained-vns constraining-vn) vn) ) )
        () ) )

    (loop (for-each-vn vn) (do
        (:= (vn:reading-vns          vn) (dreverse &&&) )
        (:= (vn:constrained-vns      vn) (dreverse &&&) )
        (:= (vn:off-live-reading-vns vn) (dreverse &&&) ) ) )
    () )


;***==========================================================================
;***
;*** (DAG.NAME:TRACE-WRITE-COUNT NAME)
;***
;***==========================================================================

(defun dag.name:trace-write-count ( name )
    ([]h *dag.name:write-count* name) )


;***==========================================================================
;***
;*** (DAG.NAME:USE-VN NAME)
;***
;***==========================================================================

(defun dag.name:use-vn ( name )
    ([]h *dag.name:use-vn* name) )


;***==========================================================================
;***
;*** (DAG.REMOVE-DEAD-VNS)
;***
;*** Removes dead code from the DAG (excessive copying by the bookkeeper
;*** can make dead code).  A VN is dead if it has no readers or off-live
;*** readers.  removing a dead VN can make its operands newly dead, so
;*** we use a standard iterative algorithm to remove them all.
;***
;***==========================================================================

(defun dag.remove-dead-vns ()
    (let ( (to-do
                (loop (for-each-vn vn)
                    (when (dag.vn:dead? vn) )
                (save vn) ) ) )

        (loop (while to-do)
              (bind vn          (pop to-do)
                    operand-vns (vn:operand-vns vn) )
        (do
            (vn:delete vn)

            (loop (for operand-vn in operand-vns)
                  (when (dag.vn:dead? operand-vn) )
            (do
                (push to-do operand-vn) ) ) ) )

        (loop (for-each-vn vn) (do
            (assert (! (dag.vn:dead? vn) )
```

```
                (h vn) t "DAG: A dead VN is still left.") ) )
        () ) )


(defun dag.vn:dead? ( vn )
    (&& (vn:name vn)
        (memq (vn:type vn) '(operation copy def) )
        (! (vn:reading-vns          vn) )
        (! (vn:off-live-reading-vns vn) ) ) )


;***==========================================================================
;***
;*** (DAG.SET-COUNTS)
;***
;***==========================================================================

(defun dag.set-counts ()
    (loop (for-each-vn vn) (do
        (:= (vn:predecessors-left vn)
            (+ (length (vn:operand-vns      vn) )
               (length (vn:constraining-vns vn) ) ) )
        (:= (vn:readers-left vn)
            (+ (length (vn:reading-vns          vn) )
               (length (vn:off-live-reading-vns vn) ) ) ) ) )
    () )


;***==========================================================================
;***
;*** (DAG.PRINT [DEPTH-FIRST?])
;***
;*** Dumps out the dag in pretty debugging format; if DEPTH-FIRST? then
;*** the dag is printed out in a depth-first order.   If DEPTH-FIRST? is a VN
;*** then only the VN and its descendents are printed.
;***
;***==========================================================================

(declare (special
    *dag.visited-vns*
    ) )

(defun dag.print (&optional depth-first?)
    (let ( (*dag.visited-vns*   *vn-set.empty-set*) )

        (if depth-first? (then
            (if (vn:is depth-first?) (then
                (dag.vn:depth-first-print depth-first?) )
            (else
                (loop (for vn in *ls.entry-vns*) (do
                    (dag.vn:depth-first-print vn) ) ) ) ) )
        (else
            (loop (for-each-vn vn) (do
                (vn:print vn) ) ) ) )
        () ) )


(defun dag.vn:depth-first-print ( vn )
    (if (! (vn-set:member? *dag.visited-vns* vn) ) (then
        (:= *dag.visited-vns* (vn-set:union1 &&& vn) )

        (vn:print vn)
```

```
        (loop (for reading-vn in (vn:reading-vns vn) ) (do
            (dag.vn:depth-first-print reading-vn) ) )
        () ) ) )


;***==========================================================================
;***
;*** (DAG.CHECK-CONSISTENCY)
;***
;*** Checks to make sure that the DAG is consistently constructed (all
;*** double links properly maintained, correct entrance and exit nodes,
;*** etc).
;***
;***==========================================================================

(defun dag.check-consistency ()
    (loop (for vn in *ls.entry-vns*) (do
        (assert (&& (== 'def (vn:type vn) )
                    (! (vn:operand-vns      vn) )
                    (! (vn:constraining-vns vn) ) )
                (h vn) ) ) )

    (loop (for vn in *ls.exit-vns*) (do
        (assert (&& (|| (== 'use (vn:type vn) )
                        (! (vn:name vn) ) )
                    (! (vn:reading-vns     vn) )
                    (! (vn:constrained-vns vn) ) )
                (h vn) ) ) )

    (loop (for-each-vn vn) (do

        (if (&& (! (vn:operand-vns      vn) )
                (! (vn:constraining-vns vn) ) )
        (then
            (assert (&& (memq vn *ls.entry-vns*)
                        (== 'def (vn:type vn) ) )
                    (h vn) ) ) )
        (loop (for operand-vn in (vn:operand-vns vn) ) (do
            (assert (memq vn (vn:reading-vns operand-vn) )
                    (h vn) t (h operand-vn) ) ) )

        (loop (for constraining-vn in (vn:constraining-vns vn) ) (do
            (assert (memq vn (vn:constrained-vns constraining-vn) )
                    (h vn) t (h constraining-vn) ) ) )

        (loop (for off-live-vn in (vn:operand-vns vn) ) (do
            (assert (memq vn (vn:off-live-reading-vns off-live-vn) )
                    (h vn) t (h off-live-vn) ) ) )

        (if (&& (! (vn:reading-vns     vn) )
                (! (vn:constrained-vns vn) ) )
        (then
            (assert (&& (memq vn *ls.exit-vns*)
                        (== 'use (vn:type vn) ) )
                    (h vn) ) ) )
        (loop (for reading-vn in (vn:reading-vns vn) ) (do
            (assert (memq vn (vn:operand-vns  reading-vn) )
                    (h vn) t (h reading-vn) ) ) )

        (loop (for constrained-vn in (vn:constrained-vns vn) ) (do
            (assert (memq vn (vn:constraining-vns constrained-vn) )
                    (h vn) t (h constrained-vn) ) ) )
```

```
        (loop (for off-live-reading-vn in (vn:off-live-reading-vns vn) ) (do
            (assert (memq vn (vn:off-live-vns off-live-reading-vn) )
                    (h vn) t (h off-live-reading-vn) ) ) ) ) )
    () )


;***==========================================================================
;***
;*** (CYCLE:DELAYS:CONSTRAINED-CYCLE CYCLE PD SD CD)
;***
;***==========================================================================

(defun cycle:delays:constrained-cycle ( cycle pd sd cd )
    (+ cycle (+ cd (max 0 (- pd sd) ) ) ) )


;***==========================================================================
;***
;*** (CYCLE:DELAYS:CONSTRAINING-CYCLE CYCLE PD SD CD)
;***
;***==========================================================================

(defun cycle:delays:constraining-cycle ( cycle pd sd cd )
    (- cycle (+ cd (max 0 (- pd sd) ) ) ) )
```

```
;***---------------------------------------------------------------------********
;*** Copyright (C) 1988 John R. Ellis.  This file may be used for      ********
;*** educational and research purposes only, by the faculty, students, ********
;*** and staff of Yale University only.                                ********
;***---------------------------------------------------------------------********
;=================================================================================
;
; This file should be included at compile time by all modules of
; LIST-SCHEDULER.
;
;=================================================================================

(declare (special
    *hash-table.not-found*

        ;*** MACHINE-MODEL
        ;***
    *ls.me-universe*
    *ls.register-bank-mes*
    *ls.constant-generator-mes*
    *ls.functional-unit-mes*

        ;*** VN
        ;***
    *ls.vn-universe*
    *vn-set.empty-set*

        ;*** DAG
        ;***
    *ls.entry-vns*
    *ls.exit-vns*
    *ls.new-defs-allowed?*
    *ls.loop-trace?*

        ;*** SCHEDULER
        ;***
    *ls.max-schedule-size*
    *ls.schedule*
    *ls.last-cycle*
    *ls.cycle*

        ;*** Options
        ;***
    *ls.immediate-constant-action*
    *ls.memory-constant-action*
    *ls.trace-information?*
    *ls.timed-functions*
    ) )


(eval-when (compile)
    (build '(
        utilities:visible-fields
        utilities:sharp-sharp
        utilities:bit-set
        utilities:object-universe
        utilities:array
        utilities:loop-min-max

        interpreter:naddr

        list-scheduler:machine-model
        list-scheduler:vn
```

```
        ) ) )
```

```
;***---------------------------------------------------------------********
;*** Copyright (C) 1983 John R. Ellis.  This file may be used for  ********
;*** educational and research purposes only, by the faculty, students, ********
;*** and staff of Yale University only.                             ********
;***---------------------------------------------------------------********
;=======================================================================
;
; The ELI model used by Rutt's CG experiments.
;
;
;
;=======================================================================

(alias '^^ 'atomconcat)

(defmacro def-bus ( me1 me2 number-of-values )
    (let ( (resource-name (atomconcat me1 '<-> me2) ) )
        '(eval-when (eval load)
            (def-connection ,me1 ,me2 ( (,resource-name) ) )
            (def-connection ,me2 ,me1 ( (,resource-name) ) )
            (def-resource-class ,resource-name ,number-of-values) ) ) )


(defun em.cluster ( i (unique-name unique-fields) )
    (let ( (ci   (atomconcat 'c i) )
           (ci-3 (atomconcat 'c (mod (- i 3) 8) ) )
           (ci-1 (atomconcat 'c (mod (- i 1) 8) ) )
           (ci+1 (atomconcat 'c (mod (+ i 1) 8) ) )
           (ci+3 (atomconcat 'c (mod (+ i 3) 8) ) ) )
        '(

                ;*** Register bank CiR
                ;***
            (def-me (name              ,(^^ ci 'r) )
                    (type              register-bank)
                    (size              64)
                    (inputs            (,(^^ ci-3 'r)
                                        ,(^^ ci-1 'r)
                                        ,(^^ ci   'r)
                                        ,(^^ ci+1 'r)
                                        ,(^^ ci+3 'r)
                                        ,(^^ ci   'c)
                                        ,(^^ ci   'm)
                                        ,(^^ ci   '+)
                                        ,(^^ ci   '=)
                                        ,.(if unique-name
                                              '(,(^^ ci unique-name) )
                                              () ) ) )
                    (read-ports       6)
                    (write-ports      3)
                    (read-resources  ( (,(^^ ci 'r-read) ) ) )
                    (write-resources ( (,(^^ ci 'r-write) ) ) ) )
            (def-resource-class ,(^^ ci 'r-read)  6)
            (def-resource-class ,(^^ ci 'r-write) 3)

                ;*** Integer adder Ci+
                ;***
            (def-me (name       ,(^^ ci '+) )
                    (type       functional-unit)
                    (delay      0)
                    (inputs     (,(^^ ci 'r)
                                 ,(^^ ci 'c) ) )
                    (resources ( (,(^^ ci '+) ) ) )
                    (operators (float fix inot idiv isub ieq imax imin
```

```
                             iadd ineg ior ige ilt ile ine iexp imul igt
                             iand iabs bitrev iland ilor) ) )
        (def-resource-class ,(^^ ci '+) 1)

            ;*** Integer tester Ci=
            ;***
        (def-me (name       ,(^^ ci '=) )
                (type       functional-unit)
                (delay      0)
                (inputs     (,(^^ ci 'r)
                             ,(^^ ci 'c) ) )
                (resources ( (,(^^ ci '=) ) ) )
                (operators (if-true if-false if-ilt if-igt if-ieq if-ine
                             if-ile if-ige if-ieq if-ine if-ile if-ige)))
        (def-resource-class ,(^^ ci '=) 1)

            ;*** Memory CiM
            ;***
        (def-me (name       ,(^^ ci 'm) )
                (type       functional-unit)
                (delay      3)
                (inputs     (,(^^ ci 'r)
                             ,(^^ ci 'c) ) )
                (resources ( (,(^^ ci 'm) ) ) )
                (operators (vbase ivload fvload ipload fpload ivstore
                             fvstore ipstore fpstore) ) )
        (def-resource-class ,(^^ ci 'm) 1)

            ;*** Constant generator CiC
            ;***
        (def-me (name       ,(^^ ci 'c) )
                (type       constant-generator)
                (resources ( (,(^^ ci 'c) ) ) )
                (constraint-function em.immediate-constant?) )
        (def-resource-class ,(^^ ci 'c) 1)


            ;*** Unique functional-unit
            ;***
        ,.(if unique-name (then
            '( (def-me (name       ,(^^ ci unique-name) )
                       (type       functional-unit)
                       (inputs     (,(^^ ci 'r)
                                    ,(^^ ci 'c) ) )
                       (resources ( (,(^^ ci unique-name) ) ) )
                       ,.unique-fields)
                (def-resource-class ,(^^ ci unique-name) 1) ) )
          (else
            () ) )

            ;*** Bus connections
            ;***
        (def-bus ,(^^ ci-3 'r) ,(^^ ci 'r) 1)
        (def-bus ,(^^ ci-1 'r) ,(^^ ci 'r) 1)
        ) ) )


(:= em.floating-test
    '(f=
      ( (delay     0)
        (operators (if-flt if-fgt if-feq if-fne if-fle if-fge
                    flt    fgt    feq    fne    fle    fge) ) ) ) )
```

```
(:= em.floating-multiply
    '(f*
      ( (delay      3)
        (operators (fmul imul) ) ) ) )

(:= em.floating-add
    '(f+
      ( (delay      2)
        (operators (fsub fadd fneg fmin fmax fabs fdiv cos sin sqrt flt fgt
                   feq fne fle fge frecip frsqrt) ) ) ) )

(defun em.immediate-constant? ( constant )
    (?( (inump constant)
        (&& (< constant 2047)
            (> constant -2048) ) )
      ( (= 0 constant)
        t)
      ( (consp constant)
        (== 'address (car constant) ) )
      ( t
        () ) ) )


(defun load-constant? ( constant )
    (if (em.immediate-constant? constant) (then
        (!== 'load *ls.immediate-constant-action*) )
    (else
        t) ) )


(eval '(def-machine-model

    ,,(em.cluster 0 em.floating-add)
    ,,(em.cluster 1 em.floating-add)
    ,,(em.cluster 2 em.floating-multiply)
    ,,(em.cluster 3 em.floating-add)
    ,,(em.cluster 4 em.floating-add)
    ,,(em.cluster 5 () )
    ,,(em.cluster 6 em.floating-multiply)
    ,,(em.cluster 7 em.floating-add)

    ) )
```

```
;***-------------------------------------------------------------------********
;*** Copyright (C) 1983 John R. Ellis.  This file may be used for      ********
;*** educational and research purposes only, by the faculty, students, ********
;*** and staff of Yale University only.                                ********
;***-------------------------------------------------------------------********
;==============================================================================
;
; GENERATE CODE HOOK
;
; This module provides a hook to save a snapshot of a trace and the
; disambiguator information in a file, so that we can excercise the code
; generator without needing to run the whole compiler.
;
; Load this file into the compiler running the ideal code generator.
; The *TR.GENERATE-CODE-HOOK* will automatically be set.   A breakpoint
; will occur during each trace.   To save that trace in a file do:
;
;   (SAVE-TRACE name)
;
; A snapshot of the trace will be saved in the file name-n.TRACE, where "n"
; is the number of the trace.
;
; The file written looks like:
;
;   (:= name-n
;       symbol-table
;       live-before
;       source-record-list
;       live-after
;       predecessors-results)
;
; where "live-before", "source-record-list", and "live-after" are the
; arguments to GENERATE-CODE, and "predecessors-results" is a list of
; the results of PREDECESSORS called on each of the operations in the
; trace in turn (not including any DEFs or USEs); the datum handed to
; PREDECESSORS is the integer position of the operation in the trace.
; "symbol-table" is a list of tuples of the form:
;
;   (name datatype rank)
;
; This file can then be used later on with TEST-BED to simulate the
; compiler interface without needing the whole compiler present.
;
; By setting the global *GCH.AUTOMATIC?* to some name XXX, then SAVE-TRACE
; will automatically be called with argument 'XXX on each trace.
;
;==============================================================================

(declare (special
    *gch.live-before*
    *gch.source-record-list*
    *gch.live-after*
    *gch.automatic?*
    ) )

(:= *tr.generate-code-hook* 'ls.generate-code-hook)

(:= *gch.trace-number* 0)
(:= *gch.automatic?*   () )


(defun ls.generate-code-hook
       ( *gch.live-before* *gch.source-record-list* *gch.live-after* )
```

```
    (if (! *gch.source-record-list*) (then
        (:= *gch.trace-number* 0) )
    (else
        (:= *gch.trace-number* (+ &&& 1) )
        (if *gch.automatic?* (then
            (save-trace *gch.automatic?*) )
        (else
            (break-point ls.generate-code-hook) ) ) ) )
    () )


(defun save-trace ( name )
    (let ( (trace-name (atomconcat name "-" *gch.trace-number*) ) )
        (iota ( (file (stconcat trace-name ".trace") '(out newversion) ) )
            (let ( (new-source-record-list () )
                   (predecessors-list      () )
                   (first-op
                        (oper:operator (car (car *gch.source-record-list*))))
                   (last-op
                        (oper:operator
                            (car (last-elt *gch.source-record-list*) ) ) ) )

                (if (!== 'def first-op) (then
                    (push *gch.source-record-list* '( (def) () () () ) ) ) )
                (if (!== 'use last-op) (then
                    (:= *gch.source-record-list*
                        (append1 &&& '( (use) () () () ) ) ) ) )

                (start-trace)
                (loop (for (oper trace-direction datum off-live)
                           in *gch.source-record-list*)
                      (incr i from 0)
                (do
                    (push new-source-record-list
                          '(,oper ,trace-direction ,i ,off-live) )
                    (push predecessors-list
                          (if (oper:property? oper 'pseudo-op)
                              ()
                              (predecessors oper trace-direction i) ) ) )
                (result
                    (:= new-source-record-list (dreverse &&&) )
                    (:= predecessors-list      (dreverse &&&) ) ) )


                (fmsg file
                    "(:= " trace-name " '(" t t

                    "(" t
                    (e (loop (for-each-name name) (do
                        (msg "    ("
                             name " "
                             (name:datatype name) " "
                             (name:rank     name) ")" t) ) ) )
                    ")" t t

                    (h *gch.live-before*      10000 10000) t t
                    (h new-source-record-list 10000 10000) t t
                    (h *gch.live-after*       10000 10000) t t
                    (h predecessors-list      10000 10000) t t
                    ") )" t)

                (filename file) ) ) ) )
```

PS:<C.S.BULLDOG.LIST-SCHEDULER.TEST>GENERATE-CODE-HOOK.LSP.5

```
;***-----------------------------------------------------------------********
;*** Copyright (C) 1983 John R. Ellis.  This file may be used for      ********
;*** educational and research purposes only, by the faculty, students, ********
;*** and staff of Yale University only.                                ********
;***-----------------------------------------------------------------********
;=======================================================================
;
; (HEIGHTS-DEPTHS.ASSIGN)
;
; Sets the :DEPTH of each VN to be the minimum estimated time that the
; VN's operation can be scheduled, using the :DELAY of operators to
; make the estimate (and assuming infinite resources).
;
; Sets :HEIGHT to be the minimum time from an exit of the DAG.
;
;=======================================================================

(eval-when (compile load)
    (include list-scheduler:declarations) )


(defun heights-depths.assign ()
    (loop (for vn in *ls.exit-vns*) (do
        (hd.vn:assign-depth vn) ) )

    (loop (for vn in *ls.entry-vns*) (do
        (hd.vn:assign-height vn) ) )
    () )


(defun hd.vn:assign-depth ( vn )
    (if (vn:depth vn) (then
        (vn:depth vn) )

    (else
        (let ( (delay (hd.vn:estimated-delay vn) ) )
            (:= (vn:depth vn)
                (loop (for operand-vn in (vn:operand-vns vn) )
                      (reduce max 0 (+ (hd.vn:estimated-delay operand-vn)
                                       (hd.vn:assign-depth operand-vn)))))

            (:= (vn:depth vn)
                (loop (for constraining-vn    in (vn:constraining-vns    vn) )
                      (for constraining-delay in (vn:constraining-delays vn) )
                      (reduce max
                              (vn:depth vn)
                              (cycle:delays:constrained-cycle
                                    (hd.vn:assign-depth    constraining-vn)
                                    (hd.vn:estimated-delay constraining-vn)
                                    delay
                                    constraining-delay) ) ) )

            (vn:depth vn) ) ) ) )


(defun hd.vn:assign-height ( vn )
    (if (vn:height vn) (then
        (vn:height vn) )

    (else
        (let ( (delay (hd.vn:estimated-delay vn) ) )
            (:= (vn:height vn)
                (loop (for reading-vn in (vn:reading-vns vn) )
```

```
                      (reduce max 0 (+ delay
                                       (hd.vn:assign-height reading-vn)))))

            (:= (vn:height vn)
                (loop (for constrained-vn in (vn:constrained-vns vn) )
                      (reduce max
                              (vn:height vn)
                              (cycle:delays:constraining-cycle
                                    (hd.vn:assign-height constrained-vn)
                                    delay
                                    (hd.vn:estimated-delay constrained-vn)
                                    (vn:constrained-vn:delay vn constrained-vn)
                                    ) ) ) )

            (vn:height vn) ) ) ) )


(defun hd.vn:estimated-delay ( vn )
    (caseq (vn:type vn)
        ( (def defi use usei copy pseudo-op)
            0)
        ( operation
          (+ 1 (operator:min-delay (vn:operator vn) ) ) )
        ( t
          (error (list vn "HD.VN:ESTIMATED-DELAY: Case error.") ) ) ) )
```

```
;***-------------------------------------------------------------------********
;*** Copyright (C) 1988 John R. Ellis.  This file may be used for      ********
;*** educational and research purposes only, by the faculty, students, ********
;*** and staff of Yale University only.                                ********
;***-------------------------------------------------------------------********
;=============================================================================
;
; Top level driver for invoking the list-scheduling compiler.
;
;=============================================================================

(eval-when (compile load)
    (include list-scheduler:declarations) )


(declare (special
    *lsex.code*                ;*** The most recent TinyLisp code.
    *lsex.actual-params*       ;*** The most recent actual parameters.
    *lsex.unoptimized-naddr*   ;*** The NADDR straight out of TinyLisp.
    *lsex.optimized-naddr*     ;*** The flow-analysis-optimized NADDR.
    *lsex.compacted-program*   ;*** The compacted ELI program.

    *gch.automatic?*           ;*** Hack debugging
    ) )

(declare
    (lexpr time-functions)
    (lexpr options.print)
    (lexpr print-functions-times)
    )


(options.set 'timed-functions '(
    compile-tiny-lisp
    fa.analyze&optimize
    compact
        generate-code
            bug.assign-mes
            sch.schedule
    simulate
    ) )


(defun lsex ( &optional (code          *lsex.code*)
                        (actual-params *lsex.actual-params*) )
    (lsex.initialize () )

    (:= *lsex.code*          code)
    (:= *lsex.actual-params* actual-params)

    (msg 0 t (e (options.print) ) 0 t)

    (unwind-protect
        (let ()
            (if *ls.timed-functions* (then
                (apply 'time-functions *ls.timed-functions*) ) )

            (if (atom *lsex.code*) (then
                (:= *lsex.unoptimized-naddr*
                    (skex.filename:read-naddr *lsex.code*) )
                (:= *lsex.optimized-naddr*
                    (fa.analyze&optimize *lsex.unoptimized-naddr* () ) ) )
            (else
```

```
                (:= *lsex.unoptimized-naddr*
                    (compile-tiny-lisp *lsex.code*) )
                (:= *lsex.optimized-naddr*
                    (fa.analyze&optimize *lsex.unoptimized-naddr* t) ) ) )

            (:= *lsex.compacted-program*
                (mis-to-eli (compact *lsex.optimized-naddr*) ) )

            (msg 0 t)
            (simulate *lsex.compacted-program* actual-params)
            (simulator.print-execution-statistics)

            (if *ls.timed-functions* (then
                (msg 0 t (e (print-function-times) ) t) ) )

            () )

        (if *ls.timed-functions* (then
            (apply 'untime-functions *ls.timed-functions*) ) )
        (:= *gch.automatic?* () ) )

    () )


(defun lsex.re-cg ()
    (lsex.initialize t)
    (unwind-protect
        (let ()
            (if *ls.timed-functions* (then
                (apply 'time-functions *ls.timed-functions*) ) )

            (:= *lsex.compacted-program*
                (mis-to-eli (compact *lsex.optimized-naddr*) ) )

            (msg 0 t)
            (simulate *lsex.compacted-program* *lsex.actual-params*)
            (simulator.print-execution-statistics)

            (if *ls.timed-functions* (then
                (msg 0 t (e (print-function-times) ) t) ) ) )

        (if *ls.timed-functions* (then
            (apply 'untime-functions *ls.timed-functions*) ) )
        (:= *gch.automatic?* () ) )
    () )


(defun lsex.initialize ( re-cg? )
    (tr.initialize)
    (initialize-code-generator)
    (if (! re-cg?) (then
        (fa.initialize)
        (de.initialize)

        (:= *lsex.code*              () )
        (:= *lsex.actual-params*     () )
        (:= *lsex.unoptimized-naddr* () )
        (:= *lsex.optimized-naddr*   () ) ) )

    (let ( (old (gcgag t) ) )
        (gc)
        (gcgag old) )
    () )
```

```
;***--------------------------------------------------------------------********
;*** Copyright (C) 1983 John R. Ellis.  This file may be used for      ********
;*** educational and research purposes only, by the faculty, students, ********
;*** and staff of Yale University only.                                ********
;***--------------------------------------------------------------------********
;==============================================================================
;
; LIST-SCHEDULER OPTIONS
;
;==============================================================================

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


(def-option *ls.immediate-constant-action* 'retained-register list-scheduler: "
Determines the strategy for handling immediate constants.   One of:

LOAD
    The constant is always taken from the nearest constant generator.
REGISTER
    The constant is assigned one or more registers at the beginning of
    the trace.
RETAINED-REGISTER
    The constant is assigned one or more registers which are kept live
    for the entire trace.  Approximates loop-invariant motion on constants.
")


(def-option *ls.memory-constant-action* 'retained-register list-scheduler: "
Determines the strategy for handling non-immediate constants that must
be loaded from memory.  One of:

LOAD
    The constant is always loaded from memory on the current trace.
REGISTER
    The constant is assigned one or more registers at the beginning of
    the trace.
RETAINED-REGISTER
    The constant is assigned one or more registers which are kept live
    for the entire trace.  Approximates loop-invariant motion on constants.
")


(def-option *ls.trace-information?* t list-scheduler: "
If true, then various markers are left laying around in compacted code
giving the origins of the code.
")


(def-option *ls.timed-functions* () list-scheduler: "
If non-(), then this is the list of functions that will be timed during
each LSEX run.
")
```

```
;***--------------------------------------------------------------------********
;*** Copyright (C) 1983 John R. Ellis.  This file may be used for       ********
;*** educational and research purposes only, by the faculty, students,  ********
;*** and staff of Yale University only.                                 ********
;***--------------------------------------------------------------------********
;==============================================================================
;
; MACHINE MODEL
;
; This module implements the basic machine model used for the list scheduler.
; A machine consists of several machine elements (MEs) forming a connected
; graph.
;
; ME
;
;======================
(def-struct me          ;
                        ;
   name                 ; The print-name of this element.
                        ;
   number               ; A unique number for bit-sets and vector
                        ; manipulation.
                        ;
   type                 ; One or REGISTER-BANK, FUNCTIONAL-UNIT,
                        ; CONSTANT-GENERATOR.
                        ;
   inputs               ; The MEs connected to the inputs and outputs
   outputs              ; of this ME.
                        ;
   operators            ; FUNCTIONAL-UNIT: List of NADDR operators
                        ; implemented by this ME.
                        ;
   delay                ; FUNCTIONAL-UNIT: The time it takes for the fu
                        ; to complete, minus 1.  I.e. a 1-cycle integer
                        ; adder has 0 delay.
                        ;
   constraint-function  ; FUNCTIONAL-UNIT, CONSTANT-GENERATOR: A boolean
                        ; function that takes one argument, a VN, returning
                        ; true if the VN can be calculated on this ME.
                        ;
   resources            ; FUNCTIONAL-UNIT, CONSTANT-GENERATOR: The resource
                        ; request needed to use the ME.
                        ;
   read-resources       ; REGISTER-BANK: The resource requests needed
   write-resources      ; to do a read or write operation on the bank.
                        ;
   size                 ; REGISTER-BANK: The number of registers in the
                        ; bank.
                        ;
   read-ports           ; REGISTER-BANK: The numbers of read and write
   write-ports          ; ports.
                        ;
   registers-left       ; REGISTER-BANK: The number of unallocated registers.
                        ;
   values               ; REGISTER-BANK: A vector of the current contents
                        ; of the registers.
                        ;
   cycles-free          ; REGISTER-BANK: A vector giving the first cycle that
                        ; an unused register is available.
                        ;
   value-names          ; REGISTER-BANK: A vector of the current names
                        ; of the register contents.
                        ;
```

```
   avoid?               ; REGISTER-BANK: A vector of booleans;  Ith element
                        ; is true if register I should be avoided if possible
                        ; during allocation.
   )                    ;
                        ;
;======================
;
; (DEF-MACHINE-MODEL
;       (DEF-ME (NAME ...) (TYPE ...) ...)
;       (DEF-ME ...)
;       (DEF-ME ...)
;       (DEF-CONNECTION ...)
;       )
;    Defines a machine model.   Undocumented.
;
; (MACHINE-MODEL.INITIALIZE)
;    Initializes this module.
;
; (NAME:ME NAME)
;    Returns the ME that has a given name, raising an error if not found.
;
; (NAME-LIST:ME-LIST NAME-LIST)
;    Maps a list of ME names onto a corresponding list of MEs.
;
; (ME-LIST:NAME-LIST ME-LIST)
;    Returns the list of names corresponding to a list of MEs.
;
; ##ME name
;    Interactive syntax for referencing an ME by name.
;
; (ME:ME:RESOURCES ME1 ME2)
;    ME2 should be one of the outputs of ME1; returns the resource request
;    needed to move a value from the output of ME1 to the input of ME2.
;
; (ME:DATUM:OK? ME DATUM)
;    True if DATUM satisfies the :CONSTRAINT-FUNCTION of ME.   If ME is
;    a constant generator, DATUM is a constant;  otherwise DATUM is a VN.
;
; *LS.REGISTER-BANK-MES*
; *LS.FUNCTIONAL-UNIT-MES*
; *LS.CONSTANT-GENERATOR-MES*
;    Lists of the various types of MEs.
;
; (OPERATOR:FUNCTIONAL-UNIT-MES OPERATOR)
;    Returns the list of functional unit MEs that implement a NADDR operator.
;
; (OPERATOR:MIN-DELAY OPERATOR)
;    Returns the minimum delay (over all functional units) needed to compute
;    a NADDR operator.
;
; (CONSTANT:CONSTANT-GENERATOR-MES CONSTANT)
;    Returns the constant generators capable of generating CONSTANT;
;    returns () if no generator generates the constant.  Uses a hash
;    table that lives for the life of the machine model to remember
;    previous results.
;
;==============================================================================

(visible-fields me name)


(eval-when (compile load)
   (include list-scheduler:declarations) )
```

```
(declare (special
    *mm.name:me*
    *mm.me:me:resources*
    *mm.connections*
    *mm.operator:mes*
    *mm.operator:min-delay*
    *mm.constant:mes*
    ) )


(eval-when (eval compile load)
    (def-object-universe *ls.me-universe*
        (object-name  me)
        (mapping-type numbered-objects)
        (object-number-function
            (lambda ( object )
                (me:number object) ) )
        (set-object-number-function
            (lambda ( object number )
                (:= (me:number object) number) ) )
        (initial-size 100) ) )


(defun machine-model.initialize ()
    (resource.initialize)
    (shortest-path.initialize)
    (me-universe:initialize)

    (:= *mm.name:me*            (hash-table:create () () () ) )
    (:= *mm.operator:mes*       (hash-table:create () () () ) )
    (:= *mm.constant:mes*       (hash-table:create 'equalt () ) )
    (:= *mm.operator:min-delay* (hash-table:create () () 100000) )

    (:= *mm.me:me:resources*         () )
    (:= *mm.connections*             () )
    (:= *ls.register-bank-mes*       () )
    (:= *ls.constant-generator-mes*  () )
    (:= *ls.functional-unit-mes*     () )
    () )


(defun name:me ( name )
    (|| ([]h *mm.name:me* name)
        (error (list name "NAME:ME: Invalid ME name.") ) ) )


(defun name-list:me-list ( name-list )
    (loop (for name in name-list) (save
        (name:me name) ) ) )


(defun me-list:name-list ( me-list )
    (loop (for me in me-list) (save
        (me:name me) ) ) )


(defun me:me:resources ( me1 me2 )
    (?( (! me1)
        () )
      ( (! me2)
        () )
```

```
      ( t
        ([]a *mm.me:me:resources* (me:number me1) (me:number me2) ) ) ) )


(defun me:create ( me )
    (me-universe:add me)
    (:= ([]h *mm.name:me* (me:name me) ) me) )


(defun machine-model.finalize ()

        ;*** Construct the lists of the different type MEs.
        ;***
    (loop (for-each-me me) (do
        (caseq (me:type me)
            (register-bank       (push *ls.register-bank-mes*       me) )
            (constant-generator  (push *ls.constant-generator-mes*  me) )
            (functional-unit     (push *ls.functional-unit-mes*     me) )
            (t
                (error (list me "Case error.") ) ) ) ) )

        ;*** Intialize the register bank MEs.
        ;***
    (loop (for me in *ls.register-bank-mes*) (do
        (:= (me:values      me) (makevector (me:size me) ) )
        (:= (me:cycles-free me) (makevector (me:size me) ) )
        (:= (me:value-names me) (makevector (me:size me) ) )
        (:= (me:avoid?      me) (makevector (me:size me) ) )
        (:= (me:registers-left me) (me:size me) ) ) )

        ;*** Replace ME names by the actual MEs, and construct
        ;*** ME:OUTPUTS from :INPUTS.
        ;***
    (loop (for-each-me me) (do
        (:= (me:inputs  me) (name-list:me-list &&&) )
        (:= (me:outputs me) (name-list:me-list &&&) ) ) )

    (loop (for-each-me me) (do
        (loop (for input-me in (me:inputs me) ) (do
            (:= (me:outputs input-me) (unionq1 &&& me) ) ) ) ) )

    (loop (for-each-me me) (do
        (:= (me:inputs  me) (dreverse &&&) )
        (:= (me:outputs me) (dreverse &&&) ) ) )

    (loop (for-each-me me) (do
        (loop (for input-me in (me:inputs me) ) (do
            (assert (memq me (me:outputs input-me) ) ) ) )
        (loop (for output-me in (me:outputs me) ) (do
            (assert (memq me (me:inputs output-me) ) ) ) ) ) )

        ;*** Instantiate the names of resource classes within resource
        ;*** requests.
        ;***
    (loop (for-each-me me) (do
        (:= (me:resources       me) (resource-request:instantiate &&&) )
        (:= (me:read-resources  me) (resource-request:instantiate &&&) )
        (:= (me:write-resources me) (resource-request:instantiate &&&) ) ) )

        ;*** Construct the ME->ME connection-resource table.
        ;***
    (:= *mm.me:me:resources*
        (array:new '(,(me-universe:size) ,(me-universe:size) ) ) )
```

```
     (loop (for (name1 name2 resource-request) in *mm.connections*)
           (bind me1 (name:me name1)
                 me2 (name:me name2) )
     (do
          (assert (memq me2 (me:outputs me1) ) )
          (:= ([]a *mm.me:me:resources* (me:number me1) (me:number me2) )
                (resource-request:instantiate resource-request) ) ) )

          ;*** Construct the operator tables.
          ;***
     (loop (for-each-me me) (do
          (loop (for operator in (me:operators me) ) (do
                (push ([]h *mm.operator:mes* operator) me)
                (:= ([]h *mm.operator:min-delay* operator)
                    (min &&& (me:delay me) ) ) ) ) ) )

          ;*** Compute the shortest path table.
          ;***
     (shortest-path.compute)
     () )


(defmacro def-me clauses
     `(me:create (me:new
          ,.(loop (for (clause-name clause-body) in clauses) (append
                `(,clause-name ',clause-body) ) ) ) ) )


(defmacro def-machine-model me-defs
     `(eval-when (eval load)
          (machine-model.initialize)
          ,.me-defs
          (machine-model.finalize) ) )


(defmacro def-connection ( me1 me2 resource-request )
     `(push *mm.connections* '(,me1 ,me2 ,resource-request) ) )


(def-sharp-sharp me
     `(name:me ',(read) ) )


(defun me:datum:ok? ( me datum )
     (|| (! (me:constraint-function me) )
          (funcall (me:constraint-function me) datum) ) )


(defun operator:min-delay ( operator )
     ([]h *mm.operator:min-delay* operator) )


(defun operator:functional-unit-mes ( operator )
     ([]h *mm.operator:mes* operator) )


(defun constant:constant-generator-mes ( constant )
     (let ( (mes ([]h *mm.constant:mes* constant) ) )
          (if (== mes *hash-table.not-found*) (then
               (:= ([]h *mm.constant:mes* constant)
                    (loop (for me in *ls.constant-generator-mes*)
                            (when (me:datum:ok? me constant) )
                    (save me) ) ) )
```

```
          (else
               mes) ) ) )
```

```
;***-----------------------------------------------------------------********
;*** Copyright (C) 1988 John R. Ellis.  This file may be used for     ********
;*** educational and research purposes only, by the faculty, students, ********
;*** and staff of Yale University only.                               ********
;***-----------------------------------------------------------------********
;=========================================================================
;
; This module convert compacted microinstructions into ELI code.
;
; To convert a list of MI's (such as returned by the compacter) into
; ELI code:
;
;     (MIS-TO-ELI 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).
;
;=========================================================================

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


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


;***========================================================================
;***
;*** (MIS-TO-ELI MI-LIST)
;***
;***========================================================================

(defun mis-to-eli ( mi-list )
    (assert (listp mi-list) )
    (:= *mte.mis-to-do* () )

    (let ( (instr-stream () )
           (instr        () ) )

            ;*** Clear the "translated" flag of every mi, and if it has
            ;*** no predecessors, push it on our to-do stack.
            ;***
        (loop (for mi in mi-list) (do
           (:= (mi:translated-to-source mi) () )
           (if (! (mi:preds mi) ) (then
               (push *mte.mis-to-do* mi) ) ) ) )

            ;*** While there are untranslated mi's, do
            ;***     convert each one to instr
            ;***
        (loop (while *mte.mis-to-do*)
              (bind mi (pop *mte.mis-to-do*) )
              (when (! (mi:translated-to-source mi) ) )
              (bind instr (mte.mi:instr mi) )
              (when instr)
        (do
            (push instr-stream instr) ) )

        (dreverse instr-stream) ) )
```

1

```
;***=========================================================================
;***
;*** (MTE.MI:INSTR MI)
;***
;*** Translates the single mi MI into an ELI instruction.  As a side
;*** effect, it pushes on *MTE.MIS-TO-DO* the successors of this MI.  Only
;*** the labels of jumps are changed.  The new instruction is also placed
;*** in the translated-to-source field of the mi.  It is possible that
;*** MIs won't have any source (they are a result of bookkeeping) and
;*** they are totally ignored here.
;***
;***=========================================================================

(defun mte.mi:instr ( 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 the first MI.
            ;***
        (if (mi:preds mi) (then
            (push popers '(() label ,(mte.mi:instr-label mi) ) ) ) )

            ;*** Translate each of the machine operations:
            ;***
        (loop (for oper in (mi:source mi) )
            (bind (fu operator . operands) oper)
        (do
            (if (== fu 'stop) (then     ;*** Gross...
                (push popers '( () stop) ) )
            (else
                (caseq operator
                    ( ( () trace copy noop)
                        (push popers oper) )
                    (t
                        (caseq (operator:group operator)
                              (goto)

                              ( (if-compare if-boolean)
                                  (push popers
                                        '(,fu
                                          ,operator
                                          ,,operands
                                          ,(if (== (car trace-dir) 'right) (then
                                               (mte.mi:instr-label
                                                    (car succs))))
                                          ,(if (== (car trace-dir) 'left) (then
                                               (mte.mi:instr-label
                                                    (car succs))))))
                                  (pop succs)
                                  (pop trace-dir) )

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

            ;*** Push the successors on the stack.
            ;***
        (loop (for succ in (mi:succs mi) )
            (when (! (mi:translated-to-source succ) ) )
```

2

```
        (do
            (push *mte.mis-to-do* succ) ) )

            ;*** At this point, SUCCS contains one MI, the fall-through.
            ;*** If it's not the first thing on the to-do stack, then
            ;*** we need to do an explicit GOTO, because the next label
            ;*** generated (the top of the stack) isn't the one we want
            ;*** for the fall-through.
            ;***
        (if (! succs) (then
            (assert (! (mi:succs mi) ) ) )  ;*** This must be the last MI.
        (else
            (assert (== 1 (length succs) ) )
            (if (|| (! *mte.mis-to-do*)
                    (!== (car succs) (car *mte.mis-to-do*) ) )
            (then
                (push popers
                      `(() goto ,(mte.mi:instr-label (car succs) ) ) ) ) ) ) )

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


;***============================================================================
;***
;*** (mte.MI:INSTR-LABEL MI)
;***
;*** Returns the label of an MI.
;***
;***============================================================================

(defun mte.mi:instr-label ( mi )
    (atomconcat 'l (mi:number mi) ) )
```

```lisp
;***----------------------------------------------------------------********
;*** Copyright (C) 1988 John R. Ellis.  This file may be used for      ********
;*** educational and research purposes only, by the faculty, students, ********
;*** and staff of Yale University only.                                ********
;***----------------------------------------------------------------********
;=========================================================================
;
; The ELI model used by Rutt's CG experiments, modified to have
; multiple register banks per cluster with a partial crossbar.
;
;=========================================================================

(alias '^^ 'atomconcat)

(defmacro def-bus ( me-list1 me-list2 number-of-values )
    (let ( (resource-name (atomconcat (car me-list1) '<-> (car me-list2) ) ) )
        `(eval-when (eval load)
            (def-resource-class ,resource-name ,number-of-values)
            ,,(loop (for me1 in me-list1) (splice
                (loop (for me2 in me-list2) (splice
                    `( (def-connection ,me1 ,me2 ( (,resource-name) ) )
                       (def-connection ,me2 ,me1 ( (,resource-name))))))))))


(defun em.cluster ( i (unique-name unique-fields) )
    (let ( (ci   (atomconcat 'c i) )
           (ci-3 (atomconcat 'c (mod (- i 3) 8) ) )
           (ci-1 (atomconcat 'c (mod (- i 1) 8) ) )
           (ci+1 (atomconcat 'c (mod (+ i 1) 8) ) )
           (ci+3 (atomconcat 'c (mod (+ i 3) 8) ) ) )
        `(

                ;*** Register bank CiR1
                ;***
            (def-me (name               ,(^^ ci 'r1) )
                    (type               register-bank)
                    (size               16)
                    (inputs             (,(^^ ci-3 'r1)
                                         ,(^^ ci-3 'r2)
                                         ,(^^ ci-3 'r3)
                                         ,(^^ ci-1 'r1)
                                         ,(^^ ci-1 'r2)
                                         ,(^^ ci-1 'r3)
                                         ,(^^ ci   'r1)
                                         ,(^^ ci   'r2)
                                         ,(^^ ci   'r3)
                                         ,(^^ ci+1 'r1)
                                         ,(^^ ci+1 'r2)
                                         ,(^^ ci+1 'r3)
                                         ,(^^ ci+3 'r1)
                                         ,(^^ ci+3 'r2)
                                         ,(^^ ci+3 'r3)
                                         ,(^^ ci   'm) ) )
                    (read-ports         2)
                    (write-ports        1)
                    (read-resources     ( (,(^^ ci 'r1-read) ) ) )
                    (write-resources    ( (,(^^ ci 'r1-write) ) ) ) )
            (def-resource-class ,(^^ ci 'r1-read)  2)
            (def-resource-class ,(^^ ci 'r1-write) 1)

                ;*** Register bank CiR2
                ;***
            (def-me (name               ,(^^ ci 'r2) )
```

```lisp
                    (type               register-bank)
                    (size               16)
                    (inputs             (,(^^ ci-3 'r1)
                                         ,(^^ ci-3 'r2)
                                         ,(^^ ci-3 'r3)
                                         ,(^^ ci-1 'r1)
                                         ,(^^ ci-1 'r2)
                                         ,(^^ ci-1 'r3)
                                         ,(^^ ci   'r1)
                                         ,(^^ ci   'r2)
                                         ,(^^ ci   'r3)
                                         ,(^^ ci+1 'r1)
                                         ,(^^ ci+1 'r2)
                                         ,(^^ ci+1 'r3)
                                         ,(^^ ci+3 'r1)
                                         ,(^^ ci+3 'r2)
                                         ,(^^ ci+3 'r3)
                                         ,(^^ ci   '+)
                                         ,(^^ ci   'c) ) )
                    (read-ports         2)
                    (write-ports        1)
                    (read-resources     ( (,(^^ ci 'r2-read) ) ) )
                    (write-resources    ( (,(^^ ci 'r2-write) ) ) ) )
            (def-resource-class ,(^^ ci 'r2-read)  2)
            (def-resource-class ,(^^ ci 'r2-write) 1)

                ;*** Register bank CiR3
                ;***
            (def-me (name               ,(^^ ci 'r3) )
                    (type               register-bank)
                    (size               16)
                    (inputs             (,(^^ ci-3 'r1)
                                         ,(^^ ci-3 'r2)
                                         ,(^^ ci-3 'r3)
                                         ,(^^ ci-1 'r1)
                                         ,(^^ ci-1 'r2)
                                         ,(^^ ci-1 'r3)
                                         ,(^^ ci   'r1)
                                         ,(^^ ci   'r2)
                                         ,(^^ ci   'r3)
                                         ,(^^ ci+1 'r1)
                                         ,(^^ ci+1 'r2)
                                         ,(^^ ci+1 'r3)
                                         ,(^^ ci+3 'r1)
                                         ,(^^ ci+3 'r2)
                                         ,(^^ ci+3 'r3)
                                         ,,(if unique-name
                                              `(,(^^ ci unique-name) )
                                               () ) ) )
                    (read-ports         2)
                    (write-ports        1)
                    (read-resources     ( (,(^^ ci 'r3-read) ) ) )
                    (write-resources    ( (,(^^ ci 'r3-write) ) ) ) )
            (def-resource-class ,(^^ ci 'r3-read)  2)
            (def-resource-class ,(^^ ci 'r3-write) 1)

                ;*** Integer adder Ci+
                ;***
            (def-me (name      ,(^^ ci '+) )
                    (type      functional-unit)
                    (delay     0)
                    (inputs    (,(^^ ci 'r2)
                                ,(^^ ci 'r3)
```

```
                   ,(^^ ci 'c) ) )
        (resources ( (,(^^ ci '+) ) ) )
        (operators (float fix inot idiv isub ieq imax imin
                   iadd ior ige ilt ile ine iexp imul igt
                   iand iabs bitrev) ) )
(def-resource-class ,(^^ ci '+) 1)

    ;*** Integer tester Ci=
    ;***
(def-me (name      ,(^^ ci '=) )
        (type      functional-unit)
        (delay     0)
        (inputs    (,(^^ ci 'r2)
                    ,(^^ ci 'r3)
                    ,(^^ ci 'c) ) )
        (resources ( (,(^^ ci '=) ) ) )
        (operators (if-true if-false if-ilt if-igt if-ieq if-ine
                    if-ile if-ige if-ieq if-ine if-ile if-ige)))
(def-resource-class ,(^^ ci '=) 1)

    ;*** Memory CiM
    ;***
(def-me (name      ,(^^ ci 'm) )
        (type      functional-unit)
        (delay     3)
        (inputs    (,(^^ ci 'r1)
                    ,(^^ ci 'r2)
                    ,(^^ ci 'c) ) )
        (resources ( (,(^^ ci 'm) ) ) )
        (operators (vbase ivload fvload ipload fpload ivstore
                    fvstore ipstore fpstore) ) )
(def-resource-class ,(^^ ci 'm) 1)

    ;*** Constant generator CiC
    ;***
(def-me (name      ,(^^ ci 'c) )
        (type      constant-generator)
        (resources ( (,(^^ ci 'c) ) ) )
        (constraint-function em.immediate-constant?) )
(def-resource-class ,(^^ ci 'c) 1)


    ;*** Unique functional-unit
    ;***
,,(if unique-name (then
    '( (def-me (name       ,(^^ ci unique-name) )
               (type       functional-unit)
               (inputs     (,(^^ ci 'r3)      ;*** Order important!
                            ,(^^ ci 'r1)
                            ,(^^ ci 'c) ) )
               (resources ( (,(^^ ci unique-name) ) ) )
               ,,unique-fields)
       (def-resource-class ,(^^ ci unique-name) 1) ) )
  (else
    () ) )

    ;*** Bus connections
    ;***
(def-bus (,(^^ ci-3 'r1) ,(^^ ci-3 'r2) ,(^^ ci-3 'r3) )
         (,(^^ ci   'r1) ,(^^ ci   'r2) ,(^^ ci   'r3) )
         1)
(def-bus (,(^^ ci-1 'r1) ,(^^ ci-1 'r2) ,(^^ ci-1 'r3) )
         (,(^^ ci   'r1) ,(^^ ci   'r2) ,(^^ ci   'r3) )
```

```
                       1)
            ) ) )


(:= em.floating-test
    '(f=
      ( (delay     0)
        (operators (if-flt if-fgt if-feq if-fne if-fle if-fge
                           flt    fgt    feq    fne    fle    fge) ) ) ) )

(:= em.floating-multiply
    '(f*
      ( (delay     3)
        (operators (fmul imul) ) ) ) )

(:= em.floating-add
    '(f+
      ( (delay     2)
        (operators (fsub fadd fmin fmax fabs fdiv cos sin sqrt) ) ) ) )


(defun em.immediate-constant? ( constant )
    (?( (inump constant)
        (&& (< constant 2047)
            (> constant -2048) ) )
      ( (= 0 constant)
        t)
      ( (consp constant)
        (== 'address (car constant) ) )
      ( t
        () ) ) )


(defun load-constant? ( constant )
    (if (em.immediate-constant? constant) (then
        (!== 'load *ls.immediate-constant-action*) )
    (else
        t) ) )


(eval '(def-machine-model

    ,,(em.cluster 0 em.floating-test)
    ,,(em.cluster 1 em.floating-add)
    ,,(em.cluster 2 em.floating-multiply)
    ,,(em.cluster 3 em.floating-add)
    ,,(em.cluster 4 em.floating-test)
    ,,(em.cluster 5 () )
    ,,(em.cluster 6 em.floating-multiply)
    ,,(em.cluster 7 em.floating-add)

    ) )
```

```
;***-----------------------------------------------------------------********
;*** Copyright (C) 1983 John R. Ellis.  This file may be used for    ********
;*** educational and research purposes only, by the faculty, students, ********
;*** and staff of Yale University only.                              ********
;***-----------------------------------------------------------------********
;============================================================================
;
; REGISTERS
;
; This module handles the allocation and deallocaton of registers from
; a register bank.
;
; (REGISTERS.INITIALIZE)
;     Initializes all the register banks to be empty.
;
; (CYCLE:ME:REGISTER:OCCUPIED? CYCLE ME REGISTER)
;     Returns the VN occuppying a register if it is not free, () otherwise.
;
; (CYCLE:ME:VN:FREE-REGISTER? CYCLE ME VN)
;     Returns true if a ME has at least one register free at CYCLE or later.
;     Eventually, we'll want to account for the operands of VN that will
;     be freed up this cycle once it is scheduled;  but now it's ignored.
;
; (CYCLE:ME:VN:ALLOCATE-REGISTER CYCLE ME VN REQUIRED-REGISTER)
;     Allocates and returns a free register from ME for the given CYCLE.
;     The register is marked as holding VN.  If non-(), REQUIRED-REGISTER
;     is a specific register that must be allocated for VN.  An error is
;     raised if there are no free registers or if REQUIRED-REGISTER is
;     specified but not free.
;
;     When choosig a free register, we first look to see if VN is of the
;     form X := f( X ) and if X is currently residing in ME.  If so, we
;     use X's current register (constraining delays guarrantee that the
;     register will be free).
;
;     Then we look in the preferred locations of the name of VN (see below)
;     to see if there's a preferred register in bank ME.  Then we look for
;     a free register that hasn't been marked as "avoid" (see below).
;     Finally, we look for any register.
;
; (CYCLE:ME:REGISTER:DEALLOCATE CYCLE ME REGISTER)
;     Deallocates REGISTER in ME, marking it as free at time CYCLE and later.
;
; (VN:RECORD-PREFERRED-LOCATION VN)
;     VN should be already have an assigned location.  Its location is
;     recorded keyed under the name of VN; when we need to allocate a
;     register for another VN of the same name, we'll prefer registers
;     so recorded.
;
;     Also, the location of VN is marked as "avoid", so that we won't
;     allocate it for another VN unless we have to.
;
;----------------------------------------------------------------------------
;
; Implementation notes:
;
; The field ME:VALUES is a vector mapping registers onto VNs occuppying
; the registers.  ME:CYCLES-FREE gives the first cycle in which the
; register can be allocated; it is a very large number if the register
; is not free.
;
; Because CYCLE:ME:VN:FREE-REGISTER?  and CYCLE:ME:VN:ALLOCATE are always
; called with the current cycle during list-scheduling, we know that the
```

```
; CYCLE argument is monotonically increasing.  The ME:REGISTERS-LEFT gives
; only a hint about how many registers left in the current cycle (it is
; <= the actual number left in the current cycle).  So when we're checking
; to see if there are any free registers, if ME:REGISTERS-LEFT tells us
; there are registers left, we can believe it.  If ME:REGISTERS-LEFT is
; 0, then we'll have to recompute it.  Gross but relatively simple and
; efficient.
;
;============================================================================

(eval-when (compile load)
  (include list-scheduler:declarations) )


(declare (special
    *reg.name:locations*         ;*** Hash table mapping names to pairs
                                 ;*** (ME REGISTER) that are to be preferred
                                 ;*** by the register allocator.
    ) )

(defvar *reg.max-integer* 100000)


(defun registers.initialize ()
    (:= *reg.name:locations* (hash-table:create 'equalt () () ) )

    (loop (for me in *ls.register-bank-mes*) (do
        (:= (me:registers-left me) (me:size me) )
        (vector:initialize (me:values      me) () )
        (vector:initialize (me:cycles-free me) 0)
        (vector:initialize (me:value-names me) () )
        (vector:initialize (me:avoid?      me) () ) ) )
    () )


(defun cycle:me:register:occupied? ( cycle me register )
    (if (< cycle ([] (me:cycles-free me) register) ) (then
        ([] (me:values me) register) )
    (else
        () ) ) )


(defun cycle:me:vn:free-register? ( cycle me vn )
    (if (> (me:registers-left me) 0) (then
        t)
    (else
            ;*** Recompute the :REGISTERS-LEFT for this cycle.   See notes
            ;*** above.
            ;***
        (:= (me:registers-left me)
            (loop (incr i from 0 to (- (me:size me) 1) )
                  (when (>= cycle ([] (me:cycles-free me) i) ) )
            (reduce + 0 1) ) )

        (> (me:registers-left me) 0) ) ) )


(defun cycle:me:vn:allocate-register ( cycle me vn required-register )
    (let*( (values      (me:values      me) )
           (avoid?      (me:avoid?      me) )
           (cycles-free (me:cycles-free me) )

           (free-register
```

```
                (|| required-register

                        ;*** If VN is of the form X := X op Y and X is
                        ;*** currently residing in ME, use its register.
                        ;***
                    (if (!== 'operation (vn:type vn) ) (then
                        () )
                    (else
                        (loop (for operand-vn in (vn:operand-vns vn) )
                              (when (&& (== (vn:name vn) (vn:name operand-vn))
                                        (== me (vn:register-bank-me operand-vn))
                        (do
                            (let ( (register (vn:register operand-vn) ) )
                                (:= ([] cycles-free register) cycle);*** Hack.
                                (return register) ) )
                        (result () ) ) ) )

                        ;*** Try for a preferred location for the VN.
                        ;***
                    (loop (for (use-me use-register)
                               in ([]h *reg.name:locations* (vn:name vn) ) )
                          (when (&& (== me use-me)
                                     (>= cycle ([] cycles-free use-register))))
                    (do
                        (return use-register) )
                    (result () ) )

                        ;*** Try for any free register, avoiding ones marked
                        ;*** as "avoid" if possible.
                        ;***
                    (loop (incr i from 0 to (- (me:size me) i) )
                          (when (>= cycle ([] cycles-free i) ) )
                          (initial avoid-register () )
                    (do
                        (if (! ([] avoid? i) ) (then
                            (return i) ) )
                        (if (! avoid-register) (then
                            (:= avoid-register i) ) ) )
                    (result avoid-register) )

                    0) ) )

        (if (< cycle ([] cycles-free free-register) ) (then
            (error (list me "No available register.") ) ) )

        (:= ([] values      free-register) vn)
        (:= ([] cycles-free free-register) *reg.max-integer*)
        (:= (me:registers-left me) (- &&& 1) )

        free-register) )


(defun cycle:me:register:deallocate ( cycle me register )
    (:= ([] (me:cycles-free me) register) cycle)
        ;***
        ;*** Sic -- we don't touch :REGISTERS-FREE, since we could be
        ;*** freeing up a register "in the future".   See above.
    () )


(defun vn:record-preferred-location ( vn )
    (assert (vn:register vn) )
```

```
    (let ( (me       (vn:register-bank-me vn) )
           (register (vn:register vn) ) )
        (:= ([] (me:avoid? me) register) t)
        (push ([]h *reg.name:locations* (vn:name vn) ) `(,me ,register) )
        () ) )
```

```
;***-------------------------------------------------------------------********
;*** Copyright (C) 1983 John R. Ellis.  This file may be used for      ********
;*** educational and research purposes only, by the faculty, students, ********
;*** and staff of Yale University only.                                ********
;***-------------------------------------------------------------------********
;==============================================================================
;
; RESOURCES
;
; This module implements the operations on resources used in the machine
; model.   It also implements a single (global) schedule of resources.
;
; A RESOURCE-CLASS defines a set of resources that are all identical as
; far as the machine model is concerned.  Any resource in the set can
; be used in place of any other.
;
; A RESOURCE-SET is a set of unused resources in the machine (for one
; cycle).  RESOURCE-SETs are represented as bit-sets.
;
; A RESOURCE-REQUEST is a request for particular resources across
; successive cycles.  It is represented as a list of lists of
; RESOURCE-CLASSes, each sublist representing the resources requested
; for a cycle (the first sublist is for the first cycle, the second sublist
; for the second cycle...)
;
; (RESOURCE.INITIALIZE)
;     Initializes this module.
;
; (RESOURCE.INITIALIZE-SCHEDULE)
;     Re-initializes the schedule of used resources to be all available.
;
; (RESOURCE-CLASS:CREATE NAME SIZE)
;     Creates a resource class with symbolic NAME and SIZE resources.
;
; (DEF-RESOURCE-CLASS 'NAME 'SIZE)
;     Same as above, except the arguments aren't evaluated.
;
; (NAME:RESOURCE-CLASS NAME)
;     Maps a symbolic name onto the corresponding RESOURCE-CLASS, raising an
;     error if there is no such class.
;
; (RESOURCE-REQUEST:INSTANTIATE RESOURCE-REQUEST)
;     Takes a resource request consisting of symbolic names of
;     RESOURCE-CLASSes and returns a new proper request with the names
;     replaced by the corresponding RESOURCE-CLASSes.
;
; (CYCLE:RESOURCE-REQUEST:AVAILABLE? CYCLE RESOURCE-REQUEST)
;     True if RESOURCE-REQUEST can be scheduled in the given cycle.
;
; (CYCLE:RESOURCE-REQUEST:FIRST-AVAILABLE-CYCLE CYCLE RESOURCE-REQUEST)
;     Returns the first cycle after CYCLE (inclusive) in which the resource
;     request can be scheduled.
;
; (CYCLE:RESOURCE-REQUEST:SCHEDULE? CYCLE RESOURCE-REQUEST)
;     Tries to schedule the resources of RESOURCE-REQUEST beginning at
;     CYCLE, returning true if successful, false o.w.  If false, the
;     schedule is left in garbaged state; use transactions below to restore
;     the state correctly.
;
; (CYCLE:RESOURCE-REQUEST:SCHEDULE  CYCLE RESOURCE-REQUEST)
;     Same as above, but raises an error if the request can't be scheduled.
;
; (RESOURCE.START-TRANSACTION)
```

```
; (RESOURCE.COMMIT-TRANSACTION)
; (RESOURCE.ABORT-TRANSCATION)
;     These three procedures let the client tentatively schedule resource
;     requests, but then undo them if he wants.  Such a tentative attempt
;     is called a "transaction"; only one transaction may be in progress
;     at a time.  After calling RESOURCE.START-TRANSACTION, all calls
;     to CYCLE:RESOURCE-REQUEST:SCHEDULE?  and
;     CYCLE:RESOURCE-REQUEST:SCHEDULE will "save away" the old values of
;     the scheduled resources.  Calling RESOURCE.ABORT-TRANSACTION will
;     restore the resource schedule back to its state at the time of the
;     call to RESOURCE.START-TRANSACTION.  Calling
;     RESOURCE.COMMIT-TRANCTION signifies that all the resources scheduled
;     are "ok" and to forget the "saved away" old schedule.
;
; (RESOURCE.PRINT-SCHEDULE)
;     Prints the current resource schedule in a pretty form.
;
;==============================================================================


;***===========================================================================
;***
;*** RESOURCE-CLASS
;***
;***=======================;***
(def-struct resource-class ;***
    name                   ;*** The printed name of the class.
    size                   ;*** The number of resources in the class.
    position               ;*** Position in the bytevector.
    ls-failures            ;***
    bug-failures           ;*** The sum of the sizes of requests for
    )                      ;*** unavailable resources from this class;
                           ;*** separate totals for list-scheduling and
                           ;*** BUG.
;***=======================;***
;***===========================================================================

(visible-fields resource-class name size)

(eval-when (compile load)
    (include list-scheduler:declarations) )
(eval-when (compile)
    (build '(list-scheduler:byte-vector) ) )

(declare (special
    *res.all-resource-classes*
    *res.name:resource-class*
    *res.total-classes*
    *res.max-class-size*
    *res.cycle:resource-set*
    *res.transaction-in-progress?*
    *res.transaction-size*
    *res.saved-bytes*
    *res.saved-positions*
    *res.saved-cycles*
    *res.failure-type*
    ) )


(defun resource.initialize ()
    (:= *res.all-resource-classes*     () )
    (:= *res.total-classes*            0)
    (:= *res.max-class-size*           0)
```

```
    (:= *res.name:resource-class*       (hash-table:create () () () ) )
    (:= *res.cycle:resource-set*        () )
    (:= *res.transaction-in-progress?* () )
    (:= *res.transaction-size*          0)
    (:= *res.saved-bytes*               () )
    (:= *res.saved-positions*           () )
    (:= *res.saved-cycles*              () )
    (:= *res.failure-type*              'ls)
    () )


(defun resource.initialize-schedule ( failure-type )
    (:= *res.cycle:resource-set* (makevector *ls.max-schedule-size*) )
    (let ( (byte-size (ceiling (// (log (+ 1 *res.max-class-size*) )
                                   (log 2) ) ) ) )
        (loop (incr i from 0 to (- *ls.max-schedule-size* 1) ) (do
            (:= ([] *res.cycle:resource-set* i)
                (byte-vector:create *res.total-classes* byte-size)))))

    (:= *res.transaction-in-progress?* () )
    (:= *res.transaction-size*          0)
    (:= *res.saved-bytes*               (makevector 100) ) ;*** Big enough?
    (:= *res.saved-positions*           (makevector 100) ) ;*** Big enough?
    (:= *res.saved-cycles*              (makevector 100) ) ;*** Big enough?

    (assert (memq failure-type '(ls bug) ) )
    (:= *res.failure-type*              failure-type)
    () )


(defun resource-class:create ( name size )
    (let ( (resource-class
                (resource-class:new
                    name           name
                    size           size
                    position       *res.total-classes*
                    ls-failures    0
                    bug-failures   0) ) )

        (:= *res.total-classes*  (+ &&& 1) )
        (:= *res.max-class-size* (max &&& size) )

        (:= *res.all-resource-classes* (append1 &&& resource-class) )
        (:= ([]h *res.name:resource-class* name) resource-class)

        resource-class) )


(defmacro def-resource-class ( name size )
    `(resource-class:create ',name ',size) )


(defun name:resource-class ( name )
    (|| ([]h *res.name:resource-class* name)
        (error (list name " isn't a RESOURCE-CLASS.") ) ) )


(defun resource-request:instantiate ( resource-request )
    (loop (for cycle-request in resource-request) (save
        (loop (for item in cycle-request) (save
            (if (consp item)
                `(,(name:resource-class (car item) )  ,(cadr item) )
                `(,(name:resource-class item)         1) ) ) ) ) ) )
```

```
(defun resource.start-transaction ()
    (assert (! *res.transaction-in-progress?*) )
    (:= *res.transaction-in-progress?* t)
    (:= *res.transaction-size*          0)
    () )


(defun resource.commit-transaction ()
    (assert *res.transaction-in-progress?*)
    (:= *res.transaction-in-progress?* () )
    (:= *res.transaction-size*          0)
    () )


(defun resource.abort-transaction ()
    (assert *res.transaction-in-progress?*)
    (loop (decr i from (- *res.transaction-size* 1) to 0) (do
        (:= ([]b ([] *res.cycle:resource-set* ([] *res.saved-cycles* i) )
                 ([] *res.saved-positions* i) )
            ([] *res.saved-bytes* i) ) ) )

    (:= *res.transaction-in-progress?* () )
    (:= *res.transaction-size*          0)
    () )


(defun cycle:resource-request:available? ( cycle resource-request )
    (loop (incr i from cycle)
          (for cycle-request in resource-request)
          (when (! (resource-set:cycle-request:available?
                        ([] *res.cycle:resource-set* i)
                        cycle-request) ) )
    (do
        (return () ) )
    (result t) ) )


(defun cycle:resource-request:first-available-cycle
       ( first-cycle resource-request )
    (loop (incr cycle from first-cycle)
          (until (cycle:resource-request:available?
                      cycle
                      resource-request) )
    (result cycle) ) )


(defun cycle:resource-request:schedule? ( cycle resource-request )
    (loop (incr i from cycle)
          (for cycle-request in resource-request)
          (when (! (cycle:cycle-request:schedule?
                        i
                        cycle-request) ) )
    (do
        (return () ) )
    (result t) ) )


(defun cycle:resource-request:schedule ( cycle resource-request )
    (if (! (cycle:resource-request:schedule? cycle resource-request) ) (then
        (error (list cycle resource-request
                     "CYCLE:RESOURCE-REQUEST:SCHEDULE:  Couldn't schedule."))))
```

```
    () )


(defun resource-set:cycle-request:available? ( resource-set cycle-request )
    (loop (for (resource-class request-size) in cycle-request)
          (when (> request-size
                   (- (resource-class:size resource-class)
                      ([]b resource-set
                           (resource-class:position resource-class)))))
    (do
        (if (== 'ls *res.failure-type*) (then
            (:= (resource-class:ls-failures resource-class)  (+ &&& 1) ) )
        (else
            (:= (resource-class:bug-failures resource-class) (+ &&& 1) ) ) )
        (return () ) )
    (result t) ) )


(defun cycle:cycle-request:schedule? ( cycle cycle-request )

(let ( (resource-set ([] *res.cycle:resource-set* cycle) ) )

    (loop (for (resource-class request-size) in cycle-request)
          (bind old-resources
                    ([]b resource-set
                         (resource-class:position resource-class) )
                new-resources
                    (+ request-size old-resources) )
    (do
        (if (> new-resources (resource-class:size resource-class) ) (then
            (if (== 'ls *res.failure-type*) (then
                (:= (resource-class:ls-failures resource-class)  (+ &&& 1) ) )
            (else
                (:= (resource-class:bug-failures resource-class) (+ &&& 1))))
            (return () ) ) )

        (if *res.transaction-in-progress?* (then
            (:= ([] *res.saved-cycles*    *res.transaction-size*)
                cycle)
            (:= ([] *res.saved-bytes*     *res.transaction-size*)
                old-resources)
            (:= ([] *res.saved-positions* *res.transaction-size*)
                (resource-class:position resource-class) )
            (:= *res.transaction-size* (+ &&& 1) ) ) )

        (:= ([]b resource-set (resource-class:position resource-class) )
            new-resources) )
    (result t) ) ) )


(defun resource.print-schedule ()
    (msg 0)
    (loop (initial first-i 0)
          (while (< first-i *ls.max-schedule-size*) )
          (bind first-set ([] *res.cycle:resource-set* first-i) )
    (do
        (loop (incr last-i from (+ first-i 1) to (- *ls.max-schedule-size* 1))
              (while (resource-set:equal first-set
                                         ([] *res.cycle:resource-set* last-i)))
        (result
            (:= last-i (- last-i 1) )
            (if (> last-i first-i) (then
                (msg (j first-i 3) ":" (j last-i 3) " ") )
```

```
            (else
                (msg (j first-i 3) "     ") ) )
            (resource-set:print first-set)
            (msg t)
            (:= first-i (+ last-i 1) ) ) ) ) )
    () )


(defun resource-set:equal ( resource-set1 resource-set2 )
    (loop (incr i from 0 to (- *res.total-classes* 1) )
          (when (!== ([]b resource-set1 i)
                     ([]b resource-set2 i) ) )
    (do
        (return () ) )
    (result t) ) )


(defun resource-set:print ( resource-set )
    (sprin1
        '(*hprint.structure-marker* (resource-set
            ,.(loop (for resource-class in *res.all-resource-classes*)
                    (bind resources
                            ([]b resource-set
                                 (resource-class:position resource-class)))
              (splice
                (?( (== 0 resources)
                    () )
                  ( (== 1 (resource-class:size resource-class) )
                    (assert (== 1 resources) )
                    '(,(resource-class:name resource-class) ) )
                  ( t
                    '( (,(resource-class:name resource-class)
                        ,resources) ) ) ) ) ) ) ) ) )


(defun resource.print-failures ()
    (msg 0 t "Resource failures:" (t 20) (jc "LS" 5) (t 30) (jc "BUG" 5) t)
    (loop (for resource-class
               in (sort *res.all-resource-classes*
                        (f:l ( r1 r2 )
                             (> (resource-class:ls-failures r1)
                                (resource-class:ls-failures r2) ) ) ) )
          (when (|| (< 0 (resource-class:ls-failures  resource-class) )
                    (< 0 (resource-class:bug-failures resource-class) ) ) )
    (do
        (msg (resource-class:name resource-class)
             (t 20) (j (resource-class:ls-failures  resource-class) 5)
             (t 30) (j (resource-class:bug-failures resource-class) 5) t)))
    () )
```

```
;***-------------------------------------------------------------********
;*** Copyright (C) 1983 John R. Ellis.  This file may be used for     ********
;*** educational and research purposes only, by the faculty, students, ********
;*** and staff of Yale University only.                               ********
;***-------------------------------------------------------------********
(eval-when (compile load)
    (include list-scheduler:declarations) )


;***=====================================================================
;***
;*** (SCH.COPY-VN:SCHEDULE VN)
;***
;*** Attempts to schedule VN, a COPY, requeueing it if it can't be scheduled
;*** because of resource conflict.
;***
;*** If the copy has no readers left, that means its only use was an
;*** off-live use at a split which has already been scheduled above the
;*** copy; the copy is now useless and can be scheduled as a NOOP.
;***
;***=====================================================================

(defun sch.copy-vn:schedule ( vn )
    (if (== 0 (vn:readers-left vn) ) (then
        (sch.noop-vn:schedule vn) )
    (else
        (resource.start-transaction)
        (caseq (sch.copy-vn:schedule? vn)
            (success
                (resource.commit-transaction) )
            (requeue
                (resource.abort-transaction)
                (sch.vn:cycle:enqueue vn (+ *ls.cycle* 1) ) )
            (abort
                (resource.abort-transaction) )
            (t
                (error "Case error.") ) )
        () ) ) )


(defun sch.copy-vn:schedule? ( vn ) (prog ()

(let*( (operand-vn    (car (vn:operand-vns vn) ) )
       (operand-me    (sch.vn:location-me operand-vn) )
       (dest-me       () )
       (dest-register () )
       (reading-vn    (sch.vn:max-height-reading-vn vn) )
       (reading-me    (&& reading-vn
                          (bug.vn:dest-me reading-vn) ) ) )

        ;*** Try scheduling the resources needed to read the operand.
        ;***
    (if (! (cycle:resource-request:schedule?
                *ls.cycle*
                (if (== 'constant-generator (me:type operand-me) )
                    (me:resources      operand-me)
                    (me:read-resources operand-me) ) ) )
    (then
        (return 'requeue) ) )

        ;*** Find a good destination register bank, and try scheduling
        ;*** the write into it.
        ;***
```

```
    (:= dest-me (sch.source-me:dest-me:vn:pick-register-bank?
                    operand-me
                    reading-me
                    vn) )
    (if (! dest-me) (then
        (return 'requeue) ) )

        ;*** At this point, all the resources of the copy have been
        ;*** scheduled.
        ;***
        ;*** If the max height reader is a USE1 that wants a particular
        ;*** register, and that register is occupied, we'll have to splice
        ;*** a COPY between the occupant of that register and the readers
        ;*** of the occupant, making this VN dependent on completion of
        ;*** that newly spliced COPY.  Hopefully, after that COPY is
        ;*** scheduled, we'll have a free register to do this COPY.
        ;***
    (if (&& reading-vn
            (== 'use1   (vn:type reading-vn) )
            (== dest-me (vn:register-bank-me reading-vn) ) )
    (then
        (if-let ( (occupant-vn (cycle:me:register:occupied?
                                    *ls.cycle*
                                    dest-me
                                    (vn:register reading-vn) ) ) )
        (then
            (return (sch.copy-vn:blocking-vn:make-dependent vn occupant-vn)))

        (:= dest-register
            (cycle:me:vn:allocate-register
                *ls.cycle* dest-me vn (vn:register reading-vn) ) ) )
    (else
        (:= dest-register
            (cycle:me:vn:allocate-register *ls.cycle* dest-me vn () ) ) ) )

    (:= (vn:scheduled-cycle vn) *ls.cycle*)

        ;*** Mark the operand as having been read this cycle.
        ;***
    (:= (vn:read-cycle operand-vn) *ls.cycle*)

        ;*** Assign the register location to this VN, inserting
        ;*** copy VNs as necesary, and decrementing the predecessor
        ;*** counts of readers and constrained VNs;  decrement the
        ;*** read counts of the operand just read.
        ;***
    (sch.vn:me:register:cycle:assign-location
        vn
        dest-me
        dest-register
        (+ *ls.cycle* 1) )

    (sch.vn:release-operands   vn)
    (sch.vn:release-successors vn)
    (sch.vn:release-off-live   vn)

        ;*** Schedule the machine operation for the COPY.
        ;***
    (sch.copy-vn:schedule-machine-operation vn)

    (return 'success) ) ) )
```

```
;***=========================================================================
;***
;*** (SCH.COPY-VN:SCHEDULE-MACHINE-OPERATION VN)
;***
;*** Makes a machine operation for VN, a COPY, and places it on the
;*** schedule.
;***
;***=========================================================================

(defun sch.copy-vn:schedule-machine-operation ( vn )
    (let*( (operand-vn (car (vn:operand-vns vn) ) )
           (source-me  (sch.vn:location-me operand-vn) ) )

        (push ([] *ls.schedule* *ls.cycle*)
              '(,vn
                ,(me:name (vn:register-bank-me vn) )
                copy
                (,(me:name (vn:register-bank-me vn) )
                 ,(vn:register vn)
                 ,(vn:name     vn) )
                ,(if (== 'constant-generator (me:type source-me) )
                     '(,(me:name source-me)
                       ,(vn:name operand-vn) )
                     '(,(me:name source-me)
                       ,(vn:register operand-vn)
                       ,(vn:name     operand-vn) ) ) ) )
        (:= *ls.last-cycle* (max &&& *ls.cycle*) )
        () ) )


;***=========================================================================
;***
;*** (SCH.COPY-VN:BLOCKING-VN:MAKE-DEPENDENT VN BLOCKING-VN)
;***
;*** VN is a COPY that, due to a USE1, wants to put its result in a
;*** register currently occupied by BLOCKING-VN; VN can't be scheduled
;*** until the BLOCKING-VN's value is moved somewhere else.  This procedure
;*** adds in constraint edges, and possibly new COPYs, to insure that
;*** BLOCKING-VN's value will be moved before scheduling of VN is attempted
;*** again.
;***
;*** If all of BLOCKING-VN's unscheduled readers do not have VN as an
;*** ancestor, then we can just wait until all of those readers are
;*** scheduled, at which time the register needed by VN will be free.
;*** This is accomplished by adding a new constraint edge between VN and
;*** the readers.
;***
;*** But if one of BLOCKING-VN's readers has VN as an ancestor, then adding
;*** a constraint edge will produce deadlock (the reader can't be scheduled
;*** until VN is, which can't be scheduled until the reader is).  So we
;*** splice a new COPY between BLOCKING-VN and all of its unscheduled
;*** readers, constraining VN to be scheduled after the new COPY (which
;*** will have freed up the desired register).  The new COPY is enqueued
;*** for the current cycle.
;***
;*** This procedure returns 'ABORT or 'REQUEUE depending on whether VN
;*** should be requeued for the next cycle (usually it shouldn't -- see
;*** below).
;***
;***=========================================================================

(defun sch.copy-vn:blocking-vn:make-dependent ( vn blocking-vn )
```

```
(let*( (unscheduled-reading-vns
            (loop (for reading-vn in (vn:reading-vns blocking-vn) )
                  (when (! (vn:scheduled-cycle reading-vn) ) )
            (save reading-vn) ) )

       (unscheduled-off-live-reading-vns
            (loop (for reading-vn in (vn:off-live-reading-vns blocking-vn) )
                  (when (! (vn:scheduled-cycle reading-vn) ) )
            (save reading-vn) ) )

       (all-unscheduled-reading-vns
            (append unscheduled-off-live-reading-vns unscheduled-reading-vns)))

    (if (for-every (reading-vn in all-unscheduled-reading-vns)
            (! (sch.vn:vn:descendant? vn reading-vn) ) )
    (then
            ;***
            ;*** The unscheduled readers of BLOCKING-VN don't have VN
            ;*** as an ancestor.  We make VN dependent on completion of
            ;*** all these readers.

        (loop (for reading-vn in all-unscheduled-reading-vns)
              (when (|| (== 'copy      (vn:type reading-vn) )
                        (== 'operation (vn:type reading-vn) ) ) )
              (initial found-one? () )
        (do
            (push (vn:constraining-vns    vn) reading-vn)
            (push (vn:constraining-delays vn) 0)
            (:= (vn:predecessors-left vn) (+ &&& 1) )

            (push (vn:constrained-vns reading-vn) vn)
            (:= found-one? t) )
        (result
            (if found-one?
                'abort
                'requeue) ) ) )
                ;***
                ;*** If we didn't find an unscheduled COPY or OPERATION
                ;*** reader, that means they have been all scheduled and
                ;*** that one of them is a conditional jump scheduled
                ;*** this cycle.  If we just requeue for another cycle
                ;*** until the jump finishes, the desired register will
                ;*** become free.

    (else   ;***
            ;*** The unscheduled readers are dependent on VN.  We've got
            ;*** to splice a new COPY between BLOCKING-VN and the readers,
            ;*** making VN dependent on the COPY.

        (let ( (new-copy-vn
                    (vn:create (vn:new
                        type         'copy
                        name         (vn:name          blocking-vn)
                        height       (vn:height        blocking-vn)
                        cycle        (vn:register-cycle blocking-vn)
                        readers-left (length all-unscheduled-reading-vns)))))

            (vn:splice-vn blocking-vn
                          new-copy-vn
                          unscheduled-reading-vns
                          unscheduled-off-live-reading-vns)
            (:= (vn:readers-left blocking-vn)
                (+ 1 (- &&& (vn:readers-left new-copy-vn) ) ) )
```

```
            (loop (for reading-vn in unscheduled-reading-vns) (do
                                        ;*** ^ (sic)
                (if (!== reading-vn vn) (then
                    (sch.vn:dequeue reading-vn) ) )
                (:= (vn:predecessors-left reading-vn) (+ &&& 1) ) ) )

            (push (vn:constraining-vns    vn) new-copy-vn)
            (push (vn:constraining-delays vn) 0)
            (:= (vn:predecessors-left vn) (+ &&& 1) )

            (push (vn:constrained-vns new-copy-vn) vn)

            (sch.vn:cycle:enqueue new-copy-vn *ls.cycle*)
            'abort) ) ) ) )


;***===========================================================================
;***
;*** (SCH.VN:VN:DESCENDANT? VN DESCENDANT-VN)
;***
;*** Returns true if DESCENDANT-VN really is a descendent of VN.  This is
;*** implemented by recursing back up through all the predecessors of
;*** DESCENDANT-VN until either we run out of predecessors, we hit a scheduled
;*** VN, or we hit VN itself.  We know we can stop recursing when we hit a
;*** scheduled predecessor, since VN (and all its descendants) are unscheduled.
;***
;*** We might have to change this implementation to use bit sets or something
;*** as in common subexpression elimination to make it linear;  we'll see
;*** how expensive it is.
;***
;***===========================================================================

(defun sch.vn:vn:descendant? ( vn descendant-vn )
    (?( (== vn descendant-vn)
        t)
      ( (vn:scheduled-cycle descendant-vn)
        () )
      ( t
        (|| (for-some (operand-vn in (vn:operand-vns descendant-vn) )
                (sch.vn:vn:descendant? vn operand-vn) )
            (for-some (constraining-vn in (vn:constraining-vns descendant-vn))
                (sch.vn:vn:descendant? vn constraining-vn) ) ) ) ) )
```

```
;***-------------------------------------------------------------------********
;*** Copyright (C) 1983 John R. Ellis.  This file may be used for      ********
;*** educational and research purposes only, by the faculty, students, ********
;*** and staff of Yale University only.                                ********
;***-------------------------------------------------------------------********
(eval-when (compile load)
    (include list-scheduler:declarations) )


;***===========================================================================
;***
;*** (SCH.OPERATION-VN:SCHEDULE VN)
;***
;*** Attempts to schedule VN, an operation, requeueing it if it can't
;*** be scheduled for any reason.
;***
;*** If the operation has no readers left, that means its only use was
;*** an off-live use at a split which has already been scheduled above
;*** the operation; the operation is now useless and can be scheduled
;*** as a NOOP.
;***
;***===========================================================================

(defun sch.operation-vn:schedule ( vn )
    (if (&& (== 0 (vn:readers-left vn) )
            (vn:name vn) )
    (then
        (sch.noop-vn:schedule vn) )
    (else
        (resource.start-transaction)
        (if (sch.operation-vn:schedule? vn) (then
            (resource.commit-transaction) )
        (else
            (resource.abort-transaction)
            (sch.vn:cycle:enqueue vn (+ *ls.cycle* 1) ) ) )
        () ) ) )


(defun sch.operation-vn:schedule? ( vn ) (prog ()

(let ( (fu-me                (vn:me vn) )
       (unique-operand-vns (noduplesq (vn:operand-vns vn) ) )
       (dest-me              () )
       (dest-register        () ) )

        ;*** Try scheduling the functional unit resources.
        ;***
    (if (! (cycle:resource-request:schedule? *ls.cycle* (me:resources fu-me)))
    (then
        (return () ) ) )

        ;*** Try scheduling the resources needed to read the operands.
        ;***
    (if (! (for-every (operand-vn in unique-operand-vns)
                (sch.operation-operand-vn:fu-me:schedule-read?
                    operand-vn fu-me) ) )
    (then
        (return () ) ) )

        ;*** Find a good destination register bank and try to schedule
        ;*** the write of the result.
        ;***
    (if (vn:name vn) (then
```

```
        (:= dest-me (sch.source-me:dest-me:vn:pick-register-bank?
                        fu-me
                        (sch.vn:max-height-dest-me vn)
                        vn) )
        (if (! dest-me) (then
            (return () ) ) ) ) )

        ;*** At this point, we know we can schedule the operation;  all the
        ;*** resources have already been scheduled.
        ;***
        ;*** Decrement the read counts of the operands, possibly freeing
        ;*** up their registers.
        ;***
    (:= (vn:scheduled-cycle vn) *ls.cycle*)

    (sch.vn:release-operands vn)

        ;*** Allocate a register for VN.
        ;***
    (if dest-me (then
        (:= dest-register
            (cycle:me:vn:allocate-register *ls.cycle* dest-me vn () ) ) ) )

        ;*** Mark the operands as having been read this cycle.
        ;***
    (loop (for operand-vn in unique-operand-vns) (do
        (:= (vn:read-cycle operand-vn) *ls.cycle*) ) )

        ;*** Assign the register location to this VN, inserting
        ;*** copy VNs as necesary, and decrementing the predecessor
        ;*** counts of readers and constrained VNs.
        ;***
    (if dest-me (then
        (sch.vn:me:register:cycle:assign-location
            vn
            dest-me
            dest-register
            (+ *ls.cycle* (+ (me:delay fu-me) 1) ) ) ) )

    (sch.vn:release-successors vn)
    (sch.vn:release-off-live   vn)

        ;*** Place a machine operation for VN on the schedule.
        ;***
    (sch.operation-vn:schedule-machine-operation vn)

    (return t) ) ) )

;***===========================================================================
;***
;*** (SCH.OPERATION-OPERAND-VN:FU-ME:SCHEDULE-READ? VN FU-ME)
;***
;*** VN is an operand of an operation, and FU-ME is the functional unit
;*** where that operation will be computed.  This procedure attempts to
;*** schedule the resources necessary to read the value of VN and deliver
;*** it to the input of FU-ME.  There are two independent considerations:
;***
;***      1.  The value of VN may be in either a register bank or a constant
;***          generator.
;***
;***      2.  The value of VN may have already been read this cycle, so
;***          we don't need to schedule the resources for the register bank
;***          or constant generator.
```

```
;***
;*** Returns true if VN can be read this cycle, false otherwise.  As a
;*** side effect, the resources for reading VN and moving it to FU-ME
;*** are scheduled.
;***
;***==========================================================================

(defun sch.operation-operand-vn:fu-me:schedule-read? ( vn fu-me ) (prog ()
    (let ( (me (sch.vn:location-me vn) ) )

        (if (&& (!== *ls.cycle* (vn:read-cycle vn) )
                (! (cycle:resource-request:schedule?
                        *ls.cycle*
                        (if (== 'constant-generator (me:type me) )
                            (me:resources      me)
                            (me:read-resources me) ) ) ) ) )
        (then
            (return () ) ) )

        (if (! (cycle:resource-request:schedule?
                    *ls.cycle*
                    (me:me:resources me fu-me) ) )
        (then
            (return () ) ) )

        (return t) ) ) )


;***==========================================================================
;***
;*** (SCH.SOURCE-ME:DEST-ME:VN:PICK-REGISTER-BANK? SOURCE-ME DEST-ME VN)
;***
;*** Attempts to find a "good" register bank connected to the outputs
;*** of SOURCE-ME to which we can move a value from SOURCE-ME.  The
;*** register-bank ME is returned if found, otherwise () is returned.
;*** As a side effect, the resources needed to write the result into the
;*** register bank are scheduled.
;***
;*** DEST-ME is the next destination of the value after it is stored in
;*** the register bank, and VN is the COPY or OPERATION VN which needs
;*** the register bank.
;***
;*** To find a good register bank, we look at ones that:
;***
;***     - are connected to the output of the SOURCE-ME;
;***     - are on the shortest path to DEST-ME;
;***     - have a free register;
;***     - can have the result moved to them from the functional unit and
;***       written (i.e. there are free resources);
;***
;*** Of these register banks, we find the ones that will cause minimum
;*** conflicts for the readers of VN.  If we still have a choice, we'd
;*** prefer a register bank that is DEST-ME itself (in case DEST-ME is
;*** a register bank, e.g. for a USE1).
;***
;***==========================================================================

(defun sch.source-me:dest-me:vn:pick-register-bank?
       ( source-me dest-me vn )

    (let*( (write-cycle        (+ *ls.cycle*
                                  (if (== 'functional-unit (me:type source-me))
                                      (me:delay source-me)
```

```
                                      0) ) )
           (source-dest-delay (me:me:delay source-me dest-me) )

           (available-mes
                (loop (for me in (me:outputs source-me) )
                      (when (== 'register-bank (me:type me) ) )
                      (when (|| (! dest-me)
                                (== me dest-me)
                                (< (me:me:delay me dest-me)
                                   source-dest-delay)
                                (&& (== 0 source-dest-delay)
                                    (== 0 (me:me:delay me dest-me) )
                                    (== 'functional-unit
                                        (me:type dest-me) ) ) ) )
                                    ;***
                                    ;*** We allow a "sideways" move --
                                    ;*** a move that gets us no closer
                                    ;*** to the dest -- only if the dest
                                    ;*** is a FU and the the value is
                                    ;*** already in a register connected
                                    ;*** to the FU.

                      (when (cycle:me:vn:free-register? *ls.cycle* me vn) )
                      (when (cycle:resource-request:available?
                                write-cycle
                                (me:me:resources source-me me) ) )
                      (when (cycle:resource-request:available?
                                write-cycle
                                (me:write-resources me) ) )
                (save-minimums me (sch.vn:me:reading-vn-conflicts vn me))))

           (register-bank-me
                (if (memq dest-me available-mes)
                    dest-me
                    (car available-mes) ) ) )

        (if (! register-bank-me) (then
            () )
        (else
            (cycle:resource-request:schedule
                write-cycle
                (me:me:resources source-me register-bank-me) )
            (cycle:resource-request:schedule
                write-cycle
                (me:write-resources register-bank-me) )
            register-bank-me) ) ) )


;***==========================================================================
;***
;*** (SCH.OPERATION-VN:SCHEDULE-MACHINE-OPERATION VN)
;***
;*** Places the machine operation for VN, an operation, on the schedule.
;***
;***==========================================================================

(defun sch.operation-vn:schedule-machine-operation ( vn )
    (let ( (last-cycle
                (+ *ls.cycle* (me:delay (vn:me vn) ) ) )
           (operands
                (loop (for operand-vn in (vn:operand-vns vn) )
                      (bind operand-me (sch.vn:location-me operand-vn) )
                (save
```

```
                (if (== 'constant-generator (me:type operand-me) ) (then
                    `(,(me:name operand-me)
                      ,(vn:name operand-vn) ) )
                (else
                    `(,(me:name operand-me)
                      ,(vn:register operand-vn)
                      ,(vn:name     operand-vn) ) ) ) ) ) ) )
    (if (vn:register-bank-me vn) (then
        (push operands
              `(,(me:name (vn:register-bank-me vn) )
                ,(vn:register vn)
                ,(vn:name     vn) ) ) ) )

    (push ([] *ls.schedule* *ls.cycle*)
          `(,vn ,(me:name (vn:me vn) ) ,(vn:operator vn) ,.operands) )

    (loop (incr cycle from (+ *ls.cycle* 1) to last-cycle) (do
        (push ([] *ls.schedule* cycle)
              `(,vn ,(me:name (vn:me vn) ) ) ) ) )

    (:= *ls.last-cycle* (max &&& last-cycle) )
    () ) )
```

```
;***-------------------------------------------------------------********
;*** Copyright (C) 1983 John R. Ellis.  This file may be used for      ********
;*** educational and research purposes only, by the faculty, students, ********
;*** and staff of Yale University only.                                ********
;***-------------------------------------------------------------********
;========================================================================
;
; SPLITS AND JOINS
;
; This module implements the part of the list-scheduler dealing with splits
; and joins, doing the work of SCHEDULE:SPLIT and SCHEDULE:JOIN.
;
; To generate the proper DEFs and USEs at splits and joins, we compute
; of the extents (lifetimes) of VNs.  That is, for each VN, we record
; the first cycle in which its value becomes available and the last cycle
; its value is used.  We keep a vector that gives, for each cycle, the
; set of VNs that are live on entry to that cycle (whose extents include
; that cycle).
;
; Partial schedules add complications.  When we ask for the set of live
; VNs at a join, what we're really interested is the set of live VNs on
; entry to the partial schedule that will be generated to preceded the
; join.  Similarly, when we ask for the set of live VNs at a split, we're
; really interested in the VNs that are live on exit from the partial
; schedule that will inserted in the split.  So we actually compute two
; different extent vectors, one for joins and one for splits.
;
; For joins, a VN doesn't become live until the cycle after the cycle
; that wrote the value (the cycle after the end of the operation); the
; VN's value becomes dead at the end of the maximum of:
;
;     1. The maximum, over all readers, of the last cycle of the reader.
;     2. The maximum, over all off-live readers (which are jumps), of
;        the last cycle of the jump.
;
; For splits, a VN becomes live on entry to the cycle after the cycle
; in which the VN is scheduled (the cycle its operation started); the
; VN's value becomes dead at the end of the cycle which is the maximum
; of:
;
;     1.  The maximum, over all readers, of the first cycle of the reader.
;     2.  The maximum, over all off-live readers (which are jumps), of
;         the cycle after the end of the jump.
;
; The second computation accounts for the fact that a value that is
; off-live at a split remains alive until after the jump has executed,
; and should be reported alive at the split.
;
; My dissertation will have to give good clear explanations of why these
; properly report the live VNs at splits and joins.
;
; To efficiently compute the live extents of VNs, we use two auxiliary
; vectors:  GEN gives for each cycle the set of VNs whose values first
; become live at the beginning of the cycle; and KILL gives for each cycle
; the set of VNs whose values become dead after the end of the cycle.
; The LIVE vector is computed as:
;
;     LIVE[ I ] := (LIVE[ I-1 ] - KILL[ I-1 ] ) + GEN[ I ]
;
;========================================================================

(eval-when (compile load)
    (include list-scheduler:declarations) )
```

```
(declare (special
    *tr.trace-number*        ;*** For debugging only.

    *sch.cycle:split-live*
    *sch.cycle:join-live*
    ) )

;***=====================================================================
;***
;*** (SCH.COMPUTE-LIVE-DEAD)
;***
;*** Computes the live-dead information for splits and joins as described
;*** above.  This procedure should be called after all the VNs have been
;*** scheduled.
;***
;***=====================================================================

(defun sch.compute-live-dead ()
    (:= *sch.cycle:split-live*
        (sch.gen-function:kill-function:compute-live-vector
            'sch.vn:split-gen-cycle
            'sch.vn:split-kill-cycle) )
    (:= *sch.cycle:join-live*
        (sch.gen-function:kill-function:compute-live-vector
            'sch.vn:join-gen-cycle
            'sch.vn:join-kill-cycle) )
    () )


;***=====================================================================
;***
;*** (SCH.GEN-FUNCTION:KILL-FUNCTION:COMPUTE-LIVE-VECTOR
;***     LIVE GEN-FUNCTION KILL-FUNCTION)
;***
;*** This function computes the LIVE vector, as described above.
;*** GEN-FUNCTION and KILL-FUNCTION take a single argument, a VN, and
;*** return the first cycle that VN is live or the last cycle it is live.
;*** The new live vector is returned.  The GEN-FUNCTION returns () if
;*** the VN doesn't produce a value.
;***
;***=====================================================================

(defun sch.gen-function:kill-function:compute-live-vector
      ( gen-function kill-function )

    (let ( (live (makevector (+ *ls.last-cycle* 3) ) )
           (gen  (makevector (+ *ls.last-cycle* 3) ) )
           (kill (makevector (+ *ls.last-cycle* 3) ) ) )
        (vector:initialize live *vn-set.empty-set*)
        (vector:initialize gen  *vn-set.empty-set*)
        (vector:initialize kill *vn-set.empty-set*)

            ;*** Set GEN and KILL for each VN.
            ;***
        (loop (for-each-vn vn)
              (bind gen-cycle  (funcall gen-function  vn) )
              (when gen-cycle)
              (bind kill-cycle (funcall kill-function vn) )
        (do
            (:= ([] gen  gen-cycle)  (vn-set:union1 &&& vn) )
            (:= ([] kill kill-cycle) (vn-set:union1 &&& vn) ) ) )
```

```
            ;*** Compute LIVE from GEN and KILL
            ;***
        (:= ([] live 0) ([] gen 0) )
        (loop (incr i from 1 to (+ 2 *ls.last-cycle*) ) (do
            (:= ([] live i)
                (vn-set:union
                    (vn-set:difference ([] live (- i 1) )
                                       ([] kill (- i 1) ) )
                    ([] gen i) ) ) ) )

        live) )


;***============================================================================
;***
;*** (SCH.VN:SPLIT-GEN-CYCLE  VN)
;*** (SCH.VN:JOIN-GEN-CYCLE   VN)
;*** (SCH.VN:SPLIT-KILL-CYCLE VN)
;*** (SCH.VN:JOIN-KILL-CYCLE  VN)
;***
;*** These functions return the first and last cycles that a VN is live, for
;*** splits and joins.  The -GEN- functions return () if the VN doesn't produce
;*** a value.
;***
;***============================================================================

(defun sch.vn:split-gen-cycle ( vn )
    (&& (vn:register-bank-me vn)
        (caseq (vn:type vn)
            (defi
                0)
            (operation
                (+ (vn:scheduled-cycle vn) 1) )
            (copy
                (+ (vn:scheduled-cycle vn) 1) )
            (t
                () ) ) ) ) )


(defun sch.vn:join-gen-cycle ( vn )
    (&& (vn:register-bank-me vn)

        (let ( (cycle
                    (caseq (vn:type vn)
                        (defi
                            0)
                        (operation
                            (+ (vn:scheduled-cycle vn)
                                (+ 1 (me:delay (vn:me vn) ) ) ) )
                        (copy
                            (+ (vn:scheduled-cycle vn) 1) )
                        (t
                            () ) ) ) )

            (if (&& (! (vn:reading-vns vn) )
                    (> cycle (sch.vn:join-kill-cycle vn) ) )
            (then
                () )
            (else
                cycle) ) ) ) )
        ;***
        ;*** Hairiness here:  If a VN has no readers but just off-live
```

```
        ;*** readers, and all those off-live readers are scheduled in
        ;*** the same cycle in which VN's value is produced, then for
        ;*** the purposes of join-live the value of VN is never generated
        ;*** (we don't want to report its locations).


(defun sch.vn:split-kill-cycle ( vn )
    (max
        (loop (for reading-vn in (vn:reading-vns vn) )
        (reduce max 0
            (caseq (vn:type reading-vn)
                (operation (vn:scheduled-cycle reading-vn) )
                (copy      (vn:scheduled-cycle reading-vn) )
                (usei      (+ 2 *ls.last-cycle*) )
                (t
                    (error (list reading-vn "Case error."))))))

        (loop (for off-live-reading-vn in (vn:off-live-reading-vns vn) )
        (reduce max 0
            (+ (vn:scheduled-cycle off-live-reading-vn)
               (+ 1
                  (me:delay (vn:me off-live-reading-vn))))))))


(defun sch.vn:join-kill-cycle ( vn )
    (max
        (loop (for reading-vn in (vn:reading-vns vn) )
        (reduce max 0
            (caseq (vn:type reading-vn)
                (operation (+ (vn:scheduled-cycle reading-vn)
                              (me:delay (vn:me reading-vn) ) ) )
                (copy      (vn:scheduled-cycle reading-vn) )
                (usei      (+ 2 *ls.last-cycle*) )
                (t
                    (error (list reading-vn "Case error."))))))

        (loop (for off-live-reading-vn in (vn:off-live-reading-vns vn) )
        (reduce max 0
            (+ (vn:scheduled-cycle off-live-reading-vn)
               (me:delay (vn:me off-live-reading-vn))))))



;***============================================================================
;***
;*** (SCH.CYCLE:SPLIT-DEF CYCLE)
;***
;*** Constructs a DEF for a split at CYCLE (the last cycle of the jump
;*** is scheduled for CYCLE).
;***
;***============================================================================

(defun sch.cycle:split-def ( cycle )
    `(def
        ,,(if *ls.trace-information?*
              `( (%trace ,*tr.trace-number* %cycle ,(+ cycle 1) ) )
              () )

        ,,(loop (for-each-vn-set-element ([] *sch.cycle:split-live*
                                              (+ cycle 1) )
                                          vn)
          (save `(,(vn:name vn)
```

```
                   ,(me:name (vn:register-bank-me vn) )
                   ,(vn:register vn) ) ) ) ) )


;***========================================================================
;***
;*** (SCH.CYCLE:JOIN-USE CYCLE)
;***
;*** Constructs a USE for a join to the beginning of CYCLE.
;***
;***========================================================================

(defun sch.cycle:join-use ( cycle )
    '(use
        ,,(if *ls.trace-information?*
              '( (%trace ,*tr.trace-number* %cycle ,(+ cycle 1) ) )
              () )

        ,,(loop (for-each-vn-set-element ([] *sch.cycle:join-live* cycle) vn)
          (save '(,(vn:name vn)
                   ,(me:name (vn:register-bank-me vn) )
                   ,(vn:register vn) ) ) ) ) )


;***========================================================================
;***
;*** (SCH.CYCLE:SPLIT-PARTIAL-SCHEDULE SPLIT-CYCLE)
;***
;*** Returns the partial schedule for a split at SPLIT-CYCLE.  The partial
;*** schedule consists of those operations spanning the boundary between
;*** SPLIT-CYCLE and the succeeding cycle.
;***
;***========================================================================

(defun sch.cycle:split-partial-schedule ( split-cycle )
    (loop (incr cycle from (+ split-cycle 1) to *ls.max-schedule-size*)
          (bind partial-cycle
                (loop (for vn&oper in ([] *ls.schedule* cycle) )
                      (bind (vn . oper) vn&oper)
                      (when (&& (== 'operation (vn:type vn) )
                                (<= (vn:scheduled-cycle vn) split-cycle) ) )
                (save vn&oper) ) )
          (while partial-cycle)
    (save
        (if *ls.trace-information?* (then
            '( (() . (() trace ,*tr.trace-number* split ,(+ 1 split-cycle)))
                ,,partial-cycle) )
        (else
            partial-cycle) ) ) ) )


;***========================================================================
;***
;*** (SCH.CYCLE:JOIN-PARTIAL-SCHEDULE JOIN-CYCLE)
;***
;*** Returns the partial schedule for a join at JOIN-CYCLE.  The partial
;*** schedule consists of those operations spanning the boundary between
;*** JOIN-CYCLE and the previous cycle.
;***
;***========================================================================

(defun sch.cycle:join-partial-schedule ( join-cycle )
    (dreverse
```

```
        (loop (decr cycle from (- join-cycle 1) to 0)
              (bind partial-cycle
                    (loop (for vn&oper in ([] *ls.schedule* cycle) )
                          (bind (vn . oper) vn&oper)
                          (when (&& (== 'operation (vn:type vn) )
                                    (<= join-cycle
                                        (+ (vn:scheduled-cycle vn)
                                           (me:delay (vn:me vn) ) ) ) ) )
                    (save vn&oper) ) )
              (while partial-cycle)
        (save
            (if *ls.trace-information?* (then
                '( (() . (() trace ,*tr.trace-number* join ,(+ 1 join-cycle)))
                    ,,partial-cycle) )
            (else
                partial-cycle) ) ) ) ) )
```

```
;***-------------------------------------------------------------------********
;*** Copyright (C) 1983 John R. Ellis.  This file may be used for     ********
;*** educational and research purposes only, by the faculty, students, ********
;*** and staff of Yale University only.                                ********
;***-------------------------------------------------------------------********
;==============================================================================
;
; LIST SCHEDULER
;
; This module implements the main part of the list scheduler and its
; interface to the outside world:
;
; (INTIALIZE-CODE-GENERATOR)
; (GENERATE-CODE LIVE-BEFORE SOURCE-RECORD-LIST LIVE-AFTER)
; (SCHEDULE:LENGTH SCHEDULE)
; (SCHEDULE:[]     SCHEDULE I)
; (SCHEDULE:JOIN   SCHEDULE I)
; (SCHEDULE:SPLIT  SCHEDULE I JUMP-NUMBER)
;
; These procedures are all documented in
; DOCUMENATION:CODE-GEN-INTERFACE.DOC.
;
; For the implementation details, start with the body of GENERATE-CODE.
; The list scheduler proper starts with SCH.SCHEDULE below.
;
;==============================================================================

(eval-when (compile load)
    (include list-scheduler:declarations) )
(eval-when (compile)
    (build '(list-scheduler:heap) ) )

(declare (special
    *tr.trace-number*       ;*** For debugging only.
    ) )

(defvar *ls.max-schedule-size* 400)


(defun initialize-code-generator ()
    (vn.initialize)
    (dag.initialize)
    (sch.initialize) )


(defun generate-code ( live-before source-record-list live-after )
    (vn.initialize)
    (dag.make live-before source-record-list live-after)
    (bug.assign-mes)

    (msg 0 t "Last BUG cycle = "
         (loop (for-each-vn vn)
               (when (vn:bug-cycle vn) )
         (maximize
            (+ (vn:bug-cycle vn)
               (caseq (vn:type vn)
                    (operation (me:delay (vn:me vn) ) )
                    (t          0) ) ) ) )
         t)

    (sch.schedule)

    *ls.schedule*)
```

```
(defun schedule:length ( schedule )
    (if (== schedule *ls.schedule*)
        (+ *ls.last-cycle* 1)
        (length schedule) ) )


(defun schedule:[] ( schedule i )
    (let*( (vn&oper-list
                (if (== schedule *ls.schedule*)
                    ([] *ls.schedule* (- i 1) )
                    (nth-elt schedule i) ) )
           (result
                (loop (for (vn . oper) in vn&oper-list) (save
                    '(,oper ,(&& vn (vn:datum vn) ) ) ) ) ) )

        (if (&& *ls.trace-information?*
                (== schedule *ls.schedule*) )
        (then
           '( ( (() trace ,*tr.trace-number*) () )
               ,,result) )
        (else
            result) ) ) )


(defun schedule:join ( schedule i )
    (assert (== schedule *ls.schedule*) )
    (let ( (cycle (- i 1) ) )
        '(,(sch.cycle:join-use                cycle)
          ,(sch.cycle:join-partial-schedule cycle) ) ) )


(defun schedule:split ( schedule i jump-number )
    (assert (== schedule *ls.schedule*) )
    (let ( (cycle (- i 1) ) )
        '(,(sch.cycle:split-def                cycle)
          ,(sch.cycle:split-partial-schedule cycle) ) ) )


;***===========================================================================
;***
;*** (SCH.SCHEDULE)
;***
;*** This procedue is the top-level list-scheduler that actually schedules
;*** VNs on the schedule.
;***
;***===========================================================================

(declare (special
    *sch.data-ready-heap*
            ;***
            ;*** This is the list-scheduler's data-ready queue, implemented
            ;*** as a HEAP.  When a VN is scheduled, the scheduling
            ;*** procedure adds any newly-data-ready VNs to the HEAP.
            ;*** VNs are kept sorted by VN:CYCLE, the earliest cycle that
            ;*** the VN could be scheduled; within the same cycle, VNs
            ;*** of larger height will take priority.

    *sch.split-live*
    *sch.join-live*
            ;***
            ;*** From SCH-SPLITS-JOINS.
```

```
    *sch.vns-to-release-off-live*
            ;***
            ;*** VNs that are "off-live read" by a conditional jump cannot
            ;*** have their readers-left count deallocated until the start
            ;*** of the cycle AFTER the cycle containing the conditional
            ;*** jump.  This variable holds the list of the off-live VNs
            ;*** of jumps scheduled in the current cycle; after the cycle
            ;*** is scheduled we'll go over the list, releasing the
            ;*** off-live VNs.
    ) )


(defun sch.initialize ()
    (:= *ls.schedule*                   () )
    (:= *ls.last-cycle*                 () )
    (:= *ls.cycle*                      () )
    (:= *sch.data-ready-heap*           () )
    (:= *sch.vns-to-release-off-live* () )

    (:= *sch.split-live*                () )
    (:= *sch.join-live*                 () )

        ;*** Initialize support modules.
        ;***
    (registers.initialize)
    (resource.initialize-schedule 'ls)
    () )


(defun sch.schedule ()

    (sch.initialize)

        ;*** Initialize data structures.
        ;***
    (:= *ls.schedule*                   (makevector *ls.max-schedule-size*) )
    (:= *ls.last-cycle*                 -1)
    (:= *ls.cycle*                      0)
    (:= *sch.data-ready-heap*           (heap:create 'sch.vn:vn:heap-compare) )

        ;*** Initialize fields in the DAG.
        ;***
    (dag.set-counts)
    (loop (for-each-vn vn) (do
        (:= (vn:cycle vn) 0) ) )

        ;*** Mark USEi-ed registers as "avoid if possible."
        ;***
    (sch.mark-used-registers)

        ;*** Schedule all the VNs in the DAG with no predecessors (DEFs and
        ;*** pseudo-ops).   This will fill the data-ready queue with "real"
        ;*** VNs.
        ;***
    (loop (for vn in *ls.entry-vns*) (do
        (sch.vn:schedule vn) ) )

        ;*** While there are more VNs to schedule, pick one out of the
        ;*** data-ready queue and try to schedule it.
        ;***
    (loop (while (> (heap:size *sch.data-ready-heap*) 0) )
          (bind vn (heap:delete *sch.data-ready-heap*) )
```

```
    (do
        (assert (! (vn:scheduled-cycle vn) )
            (h vn) t "SCHEDULER: VN already scheduled.")

        (if (!== *ls.cycle* (vn:cycle vn) ) (then
            (sch.release-off-live) ) )
        (:= *ls.cycle* (vn:cycle vn) )

        (sch.vn:schedule vn) )
    (result
        (sch.release-off-live) ) )

        ;*** Run various debugging consistency checks.
        ;***
    (sch.check-consistency)

        ;*** Compute the live-dead information.
        ;***
    (sch.compute-live-dead)

        ;*** Sort the operations on the schedule by source order (using
        ;*** VN:NUMBER -- groan).  This overkill is only to keep jumps
        ;*** scheduled in the same cycle in source order.
        ;***
    (loop (incr cycle from 0 to *ls.last-cycle*) (do
        (:= ([] *ls.schedule* cycle)
            (sort &&& (f:l ( vn&oper1 vn&oper2 )
                         (< (vn:number (car vn&oper1) )
                            (vn:number (car vn&oper2) ) ) ) ) ) ) )
    () )


;***=========================================================================
;***
;*** (SCH.MARK-USED-REGISTERS)
;***
;*** Marks every register mentioned in a USEi as "avoid if possible".
;*** The register allocator will avoid these registers if possible, thus
;*** reducing the amount of copying we need to do.
;***
;***=========================================================================

(defun sch.mark-used-registers ()
    (loop (for vn in *ls.exit-vns*)
          (when (== 'use (vn:type vn) ) )
    (do
        (loop (for usei-vn in (vn:operand-vns vn) )
              (when (vn:register usei-vn) )
        (do
            (vn:record-preferred-location usei-vn) ) ) ) )
    () )


;***=========================================================================
;***
;*** (SCH.CHECK-CONSISTENCY)
;***
;*** Performs various debugging and consistency checks after forming a
;*** schedule.
;***
;***=========================================================================

(defun sch.check-consistency ()
```

```
        ;*** Make sure all the VNs were scheduled.
        ;***
    (loop (for-each-vn vn)
          (when (! (memq (vn:type vn) '(def use) ) ) )
    (do
        (assert (vn:scheduled-cycle vn)
            (h vn) t "SCHEDULER: VN wasn't scheduled.") ) )

        ;*** Make sure registers were freed up properly.
        ;***
    (loop (for me in *ls.register-bank-mes*) (do
        (loop (incr i from 0 to (- (me:size me) 1) )
              (bind vn (cycle:me:register:occupied?
                            (+ 1 *ls.last-cycle*) me i) )
              (when vn)
        (do
            (assert (for-some (reading-vn in (vn:reading-vns vn) )
                        (== 'use1 (vn:type reading-vn) ) )
                "Register " i " in bank " (me:name me) " containing: " t
                (h vn) t
                "wasn't freed up.") ) ) ) )

    () )


;***============================================================================
;***
;*** (SCH.VN:CYCLE:ENQUEUE VN CYCLE)
;***
;*** Places VN on the data ready queue to be scheduled no earlier than
;*** CYCLE.
;***
;***============================================================================

(defun sch.vn:cycle:enqueue ( vn cycle )
    (:= (vn:cycle      vn) (max &&& cycle) )
    (heap:insert *sch.data-ready-heap* vn)
    () )


;***============================================================================
;***
;*** (SCH.VN:DEQUEUE VN)
;***
;*** Removes VN from the data ready queue "prematurely" if it is in the
;*** queue.  If it isn't, nothing is done.  It will get requeued eventually
;*** later on.
;***
;***============================================================================

(defun sch.vn:dequeue ( vn )
    (if (== 0 (vn:predecessors-left vn) ) (then
        (heap:remove *sch.data-ready-heap* vn) ) )
    () )


;***============================================================================
;***
;*** (SCH.VN:VN:HEAP-COMPARE VN1 VN2)
;***
;*** Comparison function used by the list-scheduler priority queue.  VN1
;*** takes priority over VN2 if either its :CYCLE is earlier or, if the
```

```
;*** :CYCLEs are the same, if its :HEIGHT is larger;  or if the :HEIGHTs
;*** are the same, if :BUG-CYCLE is earlier.
;***
;***============================================================================

(defun sch.vn:vn:heap-compare ( vn1 vn2 )
    (?( (< (vn:cycle vn1) (vn:cycle vn2) )
        t)

      ( (== (vn:cycle vn1) (vn:cycle vn2) )
        (?( (> (vn:height vn1) (vn:height vn2) )
            t)
          ( (== (vn:height vn1) (vn:height vn2) )
            (if (&& (vn:bug-cycle vn1)
                    (vn:bug-cycle vn2) )
            (then
                (< (vn:bug-cycle vn1) (vn:bug-cycle vn2) ) )
            (else
                t) ) )
          ( t
            () ) ) )

      ( t
        () ) ) )


;***============================================================================
;***
;*** (SCH.VN:SCHEDULE VN)
;***
;*** This procedure tries to schedule VN by dispatching to the appropriate
;*** procedure according to VN's type.  Each such procedure tries to schedule
;*** VN in the current cycle.  If it succeeds, any new data-ready successor
;*** VNs are enqueued in the data-ready heap.  If it fails, it enqueues VN
;*** back in the heap for a later cycle.
;***
;***============================================================================

(defun sch.vn:schedule ( vn )
    (caseq (vn:type vn)
        (pseudo-op (sch.pseudo-op-vn:schedule vn) )
        (def       (sch.def-vn:schedule       vn) )
        (operation (sch.operation-vn:schedule vn) )
        (copy      (sch.copy-vn:schedule      vn) )
        (use1      (sch.use1-vn:schedule      vn) )
        (t
            (error (list vn "Case error.") ) ) ) )


;***============================================================================
;***
;*** (SCH.PSEUDO-OP-VN:SCHEDULE VN)
;***
;*** Schedules a pseudo-op VN by placing its pseudo-operation on the
;*** schedule.
;***
;***============================================================================

(defun sch.pseudo-op-vn:schedule ( vn )
    (:= (vn:scheduled-cycle vn) *ls.cycle*)

    (push ([] *ls.schedule* *ls.cycle*)
        '(,vn () ,,(vn:oper vn) ) )
```

```
    (:= *ls.last-cycle* (max &&& *ls.cycle*) )
    () )


;***====================================================================
;***
;*** (SCH.DEF-VN:SCHEDULE VN)
;***
;*** Schedules a DEF VN merely by scheduling all of it readers which are
;*** DEF1 VNs.
;***
;***====================================================================

(defun sch.def-vn:schedule ( vn )
    (:= (vn:scheduled-cycle vn) *ls.cycle*)

    (loop (for reading-vn in (vn:reading-vns vn) ) (do
        (sch.def1-vn:schedule reading-vn) ) )
    () )


;***====================================================================
;***
;*** (SCH.DEF1-VN:SCHEDULE VN)
;***
;*** VN is a DEF1.  If VN is a constant DEF, then all of its successors
;*** are released into the scheduling queue.
;***
;*** If VN is a register bank, then a register is allocated and assigned
;*** to to it, and all the successors are released into the scheduling
;*** queue.  If this is a loop trace and VN's name is written on this
;*** trace, then we assume the name is an induction variable; we set the
;*** locations of all the USE1s of the same name that don't currently
;*** have a register location to be the register just allocated to this
;*** DEF VN.  This will reduce copying of the induction variables, and
;*** any copying that must be done will be integrated into the body of
;*** the loop (presumably filling "holes" in the schedule).
;***
;***====================================================================

(defun sch.def1-vn:schedule ( vn )
    (let ( (dest-me        (vn:me    vn) )
           (dest-register () ) )

        (:= (vn:scheduled-cycle vn) *ls.cycle*)

        (if (== 'constant-generator (me:type dest-me) ) (then
            (sch.vn:splice-copies vn) )

        (else
            (:= dest-register
                (cycle:me:vn:allocate-register
                    0 dest-me vn (vn:register vn) ) )

            (if (&& *ls.loop-trace?*
                    (< 0 (dag.name:trace-write-count (vn:name vn) ) ) )
            (then
                (if-let ( (use-vn (dag.name:use-vn (vn:name vn) ) ) ) (then
                    (loop (for use1-vn in (vn:operand-vns use-vn) )
                          (when (! (vn:register use1-vn) ) )
                    (do
                        (:= (vn:register-bank-me use1-vn) dest-me)
                        (:= (vn:me                use1-vn) dest-me)
```

```
                            (:= (vn:likely-mes        use1-vn) `(,dest-me) )
                            (:= (vn:register          use1-vn) dest-register)
                            (vn:record-preferred-location use1-vn)))))))

            (sch.vn:me:register:cycle:assign-location
                vn
                dest-me
                dest-register
                0) ) )

        (sch.vn:release-successors vn)
        () ) )


;***====================================================================
;***
;*** (SCH.USE1-VN:SCHEDULE VN)
;***
;*** VN is a USE1.  It is "scheduled" just by insuring that if it requires
;*** a specific location then the value is actually there.
;***
;***====================================================================

(defun sch.use1-vn:schedule ( vn )
    (:= (vn:scheduled-cycle vn) *ls.cycle*)
    (if (vn:register-bank-me vn) (then
        (let ( (operand-vn (car (vn:operand-vns vn) ) ) )
            (assert (&& (== (vn:register-bank-me vn)
                            (vn:register-bank-me operand-vn) )
                        (== (vn:register         vn)
                            (vn:register         operand-vn) ) )
                (h vn) t
                "A value didn't end up in its USE location.") ) ) )
    () )


;***====================================================================
;***
;*** (SCH.NOOP-VN:SCHEDULE VN)
;***
;*** VN is either a COPY or an OPERATION which we've decided shouldn't
;*** produce any machine opertions.   But because the bookkeeper interface
;*** needs a representative for each source operation on the schedule, we
;*** need to schedule a NOOP machine operation.
;***
;*** Currently the only NOOPs arise from some operation that produces
;*** a value whose only use is as an off-live variable at a split.  If
;*** the operation is scheduled below the split, it can be scheduled as
;*** NOOP that doesn't consume any resources.
;***
;***====================================================================

(defun sch.noop-vn:schedule ( vn )
    (:= (vn:scheduled-cycle vn) *ls.cycle*)

    (sch.vn:release-operands   vn)
    (sch.vn:release-successors vn)
    (sch.vn:release-off-live   vn)

    (push ([] *ls.schedule* *ls.cycle*)
          `(,vn
            ()
            noop
```

```
            ,.(vn:oper vn) ) )
    (:= *ls.last-cycle* (max &&& *ls.cycle*) )
    () )


;***==========================================================================
;***
;*** (SCH.VN:RELEASE-OPERANDS VN)
;***
;*** For each operand of VN, this procedure decrements the readers-left
;*** count.  If the count reaches 0, the VN's register is deallocated.
;***
;***==========================================================================

(defun sch.vn:release-operands ( vn )
    (loop (for operand-vn in (vn:operand-vns vn) ) (do
        (sch.vn:read-release operand-vn) ) )
    () )


;***==========================================================================
;***
;*** (SCH.VN:RELEASE-OFF-LIVE VN)
;*** (SCH.RELEASE-OFF-LIVE)
;***
;*** SCH.VN:RELEASE-OFF-LIVE performs a function similar to
;*** SCH.VN:RELEASE-OPERANDS.  But the readers-left count of the off-live
;*** VNs can't be decremented until after the end of the current cycle
;*** (because an off-live VN's value must stay alive until after the jump
;*** completes).  So we just remember this VN, and at the end of this
;*** cycle SCH.RELEASE-OFF-LIVE will be called to do the actual release.
;***
;***==========================================================================

(defun sch.vn:release-off-live ( vn )
    (if (vn:off-live-vns vn) (then
        (push *sch.vns-to-release-off-live* vn) ) )
    () )


(defun sch.release-off-live ()
    (loop (for vn in *sch.vns-to-release-off-live*) (do
        (loop (for off-live-vn in (vn:off-live-vns vn) ) (do
            (sch.vn:read-release off-live-vn) ) ) ) )
    (:= *sch.vns-to-release-off-live* () )
    () )


;***==========================================================================
;***
;*** (SCH.VN:READ-RELEASE VN)
;***
;*** Decrements the readers-left count of VN, and if it reaches 0, then
;*** its register is deallocated.
;***
;*** A VN's register cannot be reused by somebody else until the maximum,
;*** over all the readers, of the last cycle of the reader, and the
;*** maximum, over all the off-live readers, of the cycle following the
;*** end of the off-live reader.  This guarrantess that the a read register
;*** is "alive" for the duration of all the operations that read it.
;***
;***==========================================================================
```

```
(defun sch.vn:read-release ( vn )

    (:= (vn:readers-left vn) (- &&& 1) )
    (if (&& (<= (vn:readers-left vn) 0)
            (vn:register-bank-me vn) )
    (then
        (let ( (cycle-free
                 (max
                     (loop (for reading-vn in (vn:reading-vns vn) )
                     (reduce max 0
                         (caseq (vn:type reading-vn)
                             (operation
                                 (+ (vn:scheduled-cycle reading-vn)
                                    (me:delay (vn:me reading-vn) ) ) )
                             (copy
                                 (vn:scheduled-cycle reading-vn) )
                             (t
                                 (error (list reading-vn "Case error."))))))

                     (loop (for off-live-reading-vn
                                in (vn:off-live-reading-vns vn) )
                     (reduce max 0
                             (+ (vn:scheduled-cycle off-live-reading-vn)
                                (+ 1
                                   (me:delay (vn:me off-live-reading-vn))))))))

            (cycle:me:register:deallocate cycle-free
                                          (vn:register-bank-me vn)
                                          (vn:register         vn) ) ) ) )
    () )


;***==========================================================================
;***
;*** (SCH.VN:RELEASE-SUCCESSORS VN)
;***
;*** Decrements the predecessors-left count of each successor of VN.  If
;*** it reaches 0, the successor is enqueued for scheduling.  A reading
;*** successor is enqueued for the cycle that the value of VN becomes
;*** available; a constrained successor is enqueued for the current cycle
;*** plus the delay of the constraint.
;***
;***==========================================================================

(defun sch.vn:release-successors ( vn )
    (let ( (available-cycle
                (caseq (vn:type vn)
                    ( (operation copy)
                        (vn:register-cycle vn) )
                    ( t
                        *ls.cycle*) ) ) )

        (loop (for reading-vn in (vn:reading-vns vn) ) (do
            (:= (vn:cycle reading-vn) (max &&& available-cycle) )
            (if (>= 0 (:= (vn:predecessors-left reading-vn) (- &&& 1) ) )
            (then
                (sch.vn:cycle:enqueue reading-vn (vn:cycle reading-vn) ) ) ) ) )

        (loop (for constrained-vn in (vn:constrained-vns vn) ) (do
            (:= (vn:cycle constrained-vn)
                (max &&&
                     (cycle:delays:constrained-cycle
                        *ls.cycle*
```

```
                        (vn:delay vn)
                        (vn:delay constrained-vn)
                        (vn:constrained-vn:delay vn constrained-vn))))
            (if (>= 0 (:= (vn:predecessors-left constrained-vn) (- &&& 1) ) )
            (then
                (sch.vn:cycle:enqueue constrained-vn
                                      (vn:cycle constrained-vn))))))
        () ) )


;***===========================================================================
;***
;*** (SCH.VN:ME:REGISTER:CYCLE:ASSIGN-LOCATION VN ME REGISTER CYCLE)
;***
;*** Assigns a register location to VN that is available at time CYCLE
;*** and later.  Updates the live-dead table to signify a new live VN
;*** starting at CYCLE.  Copies are spliced between VN and all the readers
;*** of VN that can't read its value directly.
;***
;***===========================================================================

(defun sch.vn:me:register:cycle:assign-location ( vn me register cycle )
    (:= (vn:register-bank-me vn) me)
    (:= (vn:register         vn) register)
    (:= (vn:register-cycle   vn) cycle)

    (sch.vn:splice-copies vn)
    () )


;***===========================================================================
;***
;*** (SCH.VN:SPLICE-COPIES VN)
;***
;*** Splices one or more copies between VN and those readers that can't
;*** read the current location of VN directly.  Each reader is examined
;*** in turn to see if it can read the current location of VN.  All the
;*** readers that can't are grouped according to the first ME on the path
;*** from the location of VN to the desired destination of the reader
;*** (a functional unit or register bank).  Between VN and each such group
;*** is spliced a new COPY VN.
;***
;***===========================================================================

(defun sch.vn:splice-copies ( vn )
    (let ( (first-on-path&reading-vns-table () ) )

        (loop (for reading-vn in (vn:reading-vns vn) )
              (bind first-on-path (sch.vn:reading-vn:needs-copy?
                                        vn reading-vn) )
              (when first-on-path)
        (do
            (if-let ( (me&reading-vns (assoc first-on-path
                                             first-on-path&reading-vns-table)))
            (then
                (push (cdr me&reading-vns) reading-vn) )
            (else
                (push first-on-path&reading-vns-table
                    '(,first-on-path ,reading-vn) ) ) ) ) )

        (loop (for (() . reading-vns) in first-on-path&reading-vns-table)
              (bind copy-vn (vn:create (vn:new
                                type                  'copy
```

```
                                name                  (vn:name   vn)
                                height                (vn:height vn)
                                cycle                 *ls.cycle*
                                readers-left          (length reading-vns)
                                predecessors-left 1) ) )
        (do
            (vn:splice-vn vn copy-vn reading-vns () )
            (:= (vn:readers-left vn) (+ 1 (- &&& (length reading-vns) ) ) ) ) )

        () ) )


;***===========================================================================
;***
;*** (SCH.VN:READING-VN:NEEDS-COPY? VN READING-VN)
;***
;*** Returns non-() if READING-VN can't read the value of VN from its current
;*** location, false o.w.  The non-() value returned is usually the first ME
;*** on the path from the location of VN to wherever READING-VN wants it.
;***
;***===========================================================================

(defun sch.vn:reading-vn:needs-copy? ( vn reading-vn )
    (let ( (source-me (sch.vn:location-me vn) ) )

        (?      ;*** READING-VN is an operation.   A COPY is needed if the
                ;*** location of VN is not one of the direct inputs of
                ;*** the functional unit assigned READING-VN, or if
                ;*** the current location of VN conflicts with the other
                ;*** operands of READING-VN.
                ;***
          ( (&& (== 'operation (vn:type reading-vn) )
                (|| (! (memq source-me (me:inputs (vn:me reading-vn) ) ) )
                    (sch.vn:me:reading-vn:conflict?
                        vn source-me reading-vn) ) )
            (me:me:first-on-path source-me (vn:me reading-vn) ) )

                ;*** READING-VN is a USE1.  A COPY is needed if VN's
                ;*** location is a constant generator, or if VN's register
                ;*** location is not the same as that required by the
                ;*** USE1.
                ;***
          ( (== 'use1 (vn:type reading-vn) )
            (?( (== 'constant-generator (me:type source-me) )
                (if (! (vn:register-bank-me vn) ) (then
                    t)
                (else
                    (me:me:first-on-path source-me
                                          (vn:register-bank-me reading-vn)))))

              ( (&& (vn:register-bank-me reading-vn)
                    (|| (!== source-me (vn:register-bank-me reading-vn) )
                        (!== (vn:register vn) (vn:register reading-vn))))
                (me:me:first-on-path source-me
                                      (vn:register-bank-me reading-vn) ) )

              ( t
                () ) ) )

          ( t
            () ) ) ) )
```

```
;***===============================================================================
;***
;*** (SCH.VN:ME:READING-VN-CONFLICTS VN ME)
;***
;*** Returns the number of readers of VN that conflict with using ME
;*** for the result of VN.   ME is either the destination register bank
;*** to be used for VN or else the constant generator assigned to VN.
;***
;*** A reader of VN, RVN conflicts with the choice of ME for storing VN
;*** if it isn't possible for RVN to read out all of its operands in the
;*** same cycle (e.g. because there aren't enough ports).
;***
;*** Currently, we assume that the only possible resource that could cause
;*** conflict is the number of available ports (i.e. all ports of of a
;*** register bank are connected to the identical set of destinations).
;*** So checking to see if ME conflicts with previously assigned operands
;*** is merely a matter of counting the number of ME ports required by
;*** all the operands.
;***
;***===============================================================================

(defun sch.vn:me:reading-vn-conflicts ( vn me )
    (loop (for reading-vn in (vn:reading-vns vn) )
          (when (sch.vn:me:reading-vn:conflict? vn me reading-vn) )
          (reduce + 0 1) ) )


(defun sch.vn:me:reading-vn:conflict? ( vn me reading-vn )
    (if (== 'constant-generator (me:type me) ) (then
        (loop (for operand-vn in (vn:operand-vns reading-vn) )
              (when (!== vn operand-vn) )
              (when (== me (vn:me operand-vn) ) )
        (do
            (return t) )
        (result () ) ) )

    (else
        (let ( (read-ports (me:read-ports me) ) )   ;*** Compiler bug.
            (> (sch.vn:me:reading-vn:required-ports vn me reading-vn)
               read-ports) ) ) ) )


(defun sch.vn:me:reading-vn:required-ports ( vn me reading-vn )
    (loop (for operand-vn in (vn:operand-vns reading-vn) )
          (when (!== vn operand-vn) )
          (when (== me (vn:register-bank-me operand-vn) ) )
    (reduce + 1 1) ) )


;***===============================================================================
;***
;*** (SCH.VN:MAX-HEIGHT-READING-VN VN)
;***
;*** Returns the reader of VN of maximum height.
;***
;***===============================================================================

(defun sch.vn:max-height-reading-vn ( vn )
    (loop (for reading-vn in (vn:reading-vns vn) )
          (maximize reading-vn (vn:height reading-vn) ) ) )


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

```
;***
;*** (SCH.VN:MAX-HEIGHT-DEST-ME VN)
;***
;*** Finds the reader of VN that has maximum height, and then returns
;*** the destination ME that VN must deliver its result to for it to be
;*** used by the reader.  The ME returned depends on the type of
;*** READING-VN:
;***
;***     OPERATION:   the functional unit assiged to the VN.
;***     COPY:        wherever the max height reader of VN wants its value.
;***     USE1:        any register bank assigned to the use.
;***
;*** Returning () means that we don't have any particular preference for
;*** a a destination ME; this should only occur if VN is a USE1 and doesn't
;*** have a required location.
;***
;*** GROAN:  We use an unexported function from BUG.
;***
;***===============================================================================

(defun sch.vn:max-height-dest-me ( vn )
    (bug.vn:dest-me
        (loop (for reading-vn in (vn:reading-vns vn) )
              (maximize reading-vn (vn:height reading-vn) ) ) ) )


;***===============================================================================
;***
;*** (SCH.VN:LOCATION-ME VN)
;***
;*** Returns the current ME location of the value of VN which must be
;*** read to get the value of VN.  Normally, this is the register bank
;*** ME, unless VN is a DEF1 assigned a constant generator, in which case
;*** the c.g.  ME is returned.
;***
;***===============================================================================

(defun sch.vn:location-me ( vn )
    (if (&& (vn:me vn)
            (== 'constant-generator (me:type (vn:me vn) ) ) )
        (vn:me                vn)
        (vn:register-bank-me vn) ) )



(defun sch.print-schedule ()
    (msg 0 t "Schedule:" t)
    (loop (incr i from 0 to *ls.last-cycle*) (do
        (msg (j i 8) ": "
             (h (loop (for (vn . oper) in ([] *ls.schedule* i) ) (save
                     oper) ) )
             t) ) )
    () )
```

```
;***-------------------------------------------------------------------********
;*** Copyright (C) 1988 John R. Ellis.  This file may be used for      ********
;*** educational and research purposes only, by the faculty, students, ********
;*** and staff of Yale University only.                                ********
;***-------------------------------------------------------------------********
;==============================================================================
;
; A pipelined sequential model used for comparing with ELI-MODEL.LSP.
;
; One operation can be initiated every cycle, and the register port can
; read and write a value every cycle.
;
;
;==============================================================================

(def-machine-model

        ;*** Register bank R
        ;***
    (def-me (name             r)
            (type             register-bank)
            (size             #.(* 8 64) )
            (inputs           (fu1 fu3 fu4 r c) )
            (read-ports       3)
            (write-ports      1)
            (read-resources  ( (r-read) ) )
            (write-resources ( (r-write) ) ) )
    (def-resource-class r-read  3)
    (def-resource-class r-write 1)

        ;*** Constant generator C
        ;***
    (def-me (name      c)
            (type      constant-generator)
            (resources ( (constant) ) )
            (constraint-function sm.immediate-constant?) )
    (def-resource-class constant 1)

        ;*** 1-cycle functional unit FU1
        ;***
    (def-me (name      fu1)
            (type      functional-unit)
            (delay     0)
            (inputs    (r c) )
            (resources ( (cycle) ) )
            (operators (float fix inot idiv isub ieq imax imin iadd ineg ior
                        ige ilt ile ine iexp imul igt iand iabs bitrev
                        if-true if-false if-ilt if-igt if-ieq if-ine if-ile
                        if-ige if-ieq if-ine if-ile if-ige) ) )

        ;*** 3-cycle functional unit FU3
        ;***
    (def-me (name      fu3)
            (type      functional-unit)
            (delay     2)
            (inputs    (r c) )
            (resources ( (cycle) ) )
            (operators (fsub fadd fneg fmin fmax fabs fdiv cos sin sqrt flt
                        fgt feq fne fle fge) ) )

        ;*** 4-cycle functional unit FU4
        ;***
    (def-me (name      fu4)
```

```
            (type      functional-unit)
            (delay     3)
            (inputs    (r c) )
            (resources ( (cycle) ) )
            (operators (vbase ivload fvload ipload fpload ivstore fvstore
                        ipstore fpstore fmul imul) ) )

    (def-resource-class cycle 1) )

(defun sm.immediate-constant? ( constant )
    (?( (inump constant)
        (&& (< constant 2047)
            (> constant -2048) ) )
      ( (= 0 constant)
        t)
      ( (consp constant)
        (== 'address (car constant) ) )
      ( t
        () ) ) )


(defun load-constant? ( constant )
    (if (sm.immediate-constant? constant) (then
        (!== 'load *ls.immediate-constant-action*) )
    (else
        t) ) )
```

```
;***-------------------------------------------------------------------********
;*** Copyright (C) 1983 John R. Ellis.  This file may be used for       ********
;*** educational and research purposes only, by the faculty, students,  ********
;*** and staff of Yale University only.                                 ********
;***-------------------------------------------------------------------********
;============================================================================
;
; The ELI model used by Rutt's CG experiments.
;
;
;
;============================================================================

(alias '^^ 'atomconcat)

(defun em.cluster ( i (unique-name unique-fields) )
    (let ( (ci   (atomconcat 'c i) )
           (ci-3 (atomconcat 'c (mod (- i 3) 8) ) )
           (ci-1 (atomconcat 'c (mod (- i 1) 8) ) )
           (ci+1 (atomconcat 'c (mod (+ i 1) 8) ) )
           (ci+3 (atomconcat 'c (mod (+ i 3) 8) ) ) )
        '(

            ;*** Register bank CiR
            ;***
          (def-me (name              ,(^^ ci 'r) )
                  (type              register-bank)
                  (size              64)
                  (inputs            (,(^^ ci-3 'r)
                                       ,(^^ ci-1 'r)
                                       ,(^^ ci   'r)
                                       ,(^^ ci+1 'r)
                                       ,(^^ ci+3 'r)
                                       ,(^^ ci   'c)
                                       ,(^^ ci   'm)
                                       ,(^^ ci   '+)
                                       ,(^^ ci   '=)
                                       ,(^^ ci   unique-name) ) )
                  (read-ports        6)
                  (write-ports       3)
                  (read-resources  ( (,(^^ ci 'r-read) ) ) )
                  (write-resources ( (,(^^ ci 'r-write) ) ) ) )
          (def-resource-class ,(^^ ci 'r-read)  6)
          (def-resource-class ,(^^ ci 'r-write) 3)

            ;*** Integer adder Ci+
            ;***
          (def-me (name      ,(^^ ci '+) )
                  (type      functional-unit)
                  (delay     0)
                  (inputs    (,(^^ ci 'r)
                               ,(^^ ci 'c) ) )
                  (resources ( (,(^^ ci '+) ) ) )
                  (operators (float fix inot idiv isub ieq imax imin
                               iadd ior ige ilt ile ine iexp imul igt
                               iand iabs bitrev) ) )
          (def-resource-class ,(^^ ci '+) 1)

            ;*** Integer tester Ci=
            ;***
          (def-me (name      ,(^^ ci '=) )
                  (type      functional-unit)
                  (delay     0)
                  (inputs    (,(^^ ci 'r)
```

```
                               ,(^^ ci 'c) ) )
                  (resources ( (,(^^ ci '=) ) ) )
                  (operators (if-true if-false if-ilt if-igt if-ieq if-ine
                               if-ile if-ige if-ieq if-ine if-ile if-ige)))
          (def-resource-class ,(^^ ci '=) 1)

            ;*** Memory CiM
            ;***
          (def-me (name      ,(^^ ci 'm) )
                  (type      functional-unit)
                  (delay     2)
                  (inputs    (,(^^ ci 'r)
                               ,(^^ ci 'c) ) )
                  (resources ( (,(^^ ci 'm) ) ) )
                  (operators (vbase ivload fvload ipload fpload ivstore
                               fvstore ipstore fpstore) ) )
          (def-resource-class ,(^^ ci 'm) 1)

            ;*** Constant generator CiC
            ;***
          (def-me (name      ,(^^ ci 'c) )
                  (type      constant-generator)
                  (resources ( (,(^^ ci 'c) ) ) )
                  (constraint-function
                             (lambda ( constant )
                                   (?( (inump constant)
                                       (&& (< constant 2047)
                                           (> constant -2048) ) )
                                     ( (= 0 constant)
                                       t)
                                     ( (consp constant)
                                       (== 'address (car constant) ) )
                                     ( t
                                       () ) ) ) ) ) )
          (def-resource-class ,(^^ ci 'c) 1)


            ;*** Unique functional-unit
            ;***
          (def-me (name      ,(^^ ci unique-name) )
                  (type      functional-unit)
                  (inputs    (,(^^ ci 'r)
                               ,(^^ ci 'c) ) )
                  (resources ( (,(^^ ci unique-name) ) ) )
                  ,,unique-fields)
          (def-resource-class ,(^^ ci unique-name) 1)

            ;*** Bus connections
            ;***
          (def-bus ,(^^ ci-3 'r) ,(^^ ci 'r) 1)
          (def-bus ,(^^ ci+3 'r) ,(^^ ci 'r) 1)
          (def-bus ,(^^ ci-1 'r) ,(^^ ci 'r) 1)
          (def-bus ,(^^ ci+1 'r) ,(^^ ci 'r) 1)

          ) ) )


(:= em.floating-test
     '(f=
       ( (delay     0)
         (operators (if-flt if-fgt if-feq if-fne if-fle if-fge) ) ) ) )

(:= em.table-lookup
```

```
    `(t
      ( (delay     0)
        (operators (fdiv idiv cos sin sqrt) ) ) ) )

(:= em.floating-multiply
    `(f*
      ( (delay     3)
        (operators (fmul imul) ) ) ) )

(:= em.floating-add
    `(f+
      ( (delay     2)
        (operators (fsub fadd fmin fmax fabs) ) ) ) )


(eval `(def-machine-model

    .,(em.cluster 0 em.floating-test)
    .,(em.cluster 1 em.table-lookup)
    .,(em.cluster 2 em.floating-multiply)
    .,(em.cluster 3 em.floating-add)
    .,(em.cluster 4 em.floating-test)
    .,(em.cluster 5 em.table-lookup)
    .,(em.cluster 6 em.floating-multiply)
    .,(em.cluster 7 em.floating-add)
    ) )
```

```
;***-----------------------------------------------------------------********
;*** Copyright (C) 1988 John R. Ellis.  This file may be used for    ********
;*** educational and research purposes only, by the faculty, students, ********
;*** and staff of Yale University only.                               ********
;***-----------------------------------------------------------------********
;===========================================================================
;
; SHORTEST PATH
;
; This module provides functions for constructing and accessing the
; shortest paths between machine elements.
;
; (SHORTEST-PATH.INITIALIZE)
;     Initializes this module.
;
; (SHORTEST-PATH.COMPUTE)
;     Constructs the shortest-path tables using a transitive closure
;     algorithm.
;
; (SHORTEST-PATH.PRINT)
;     Prints out the table of shortest-path delays between all MEs.
;
; (ME:ME:DELAY ME1 ME2)
;     Returns the delay in getting a value from the outputs of ME1 to
;     the inputs of ME2.  The delay is the number of cycles required for
;     the move and is currently equivalent to the number of register banks
;     along the path between the outputs of ME1 and the inputs of ME2.
;     Either ME1 or ME2 may be (), in which case the result is 0.
;
; (ME:ME:FIRST-ON-PATH ME1 ME2)
;     Returns the list of of MEs that are first on the shortest path from
;     ME1 to ME2; () if ME1 = ME2.
;
;===========================================================================

(eval-when (compile load)
    (include list-scheduler:declarations) )

(declare (special
    *sp.me:me:path*
    *sp.me:me:delay*
    ) )


(defun shortest-path.initialize ()
    (:= *sp.me:me:path*  () )
    (:= *sp.me:me:delay* () )
    () )


(defun shortest-path.compute ()
    (let*( (size (me-universe:size) )
           (path (array:new `(,size ,size) ) )
           (delay (array:new `(,size ,size) ) ) )

        (:= *sp.me:me:path*  path)
        (:= *sp.me:me:delay* delay)

        (loop (for-each-me me) (do
            (loop (for output-me in (me:outputs me) ) (do
                (:= ([]a delay (me:number me) (me:number output-me) )
                    0)
                (push ([]a path (me:number me) (me:number output-me) )
```

```
                    output-me) ) ) ) )

        (msg 0)
        (loop (incr k from 0 to (- size 1) )
              (bind me-k (me-universe:number:me k) )
              (when (== 'register-bank (me:type me-k) ) )
        (do
            (msg " " k)

            (loop (incr i from 0 to (- size 1) )
                  (when (!== i k) )
                  (bind i-k-delay ([]a delay i k) )
                  (when i-k-delay)
            (do
                (loop (incr j from 0 to (- size 1) )
                      (when (!== j k) )
                      (bind i-j-delay ([]a delay i j)
                            k-j-delay ([]a delay k j) )
                      (when k-j-delay)
                      (bind i-k-j-delay (+ 1 (+ i-k-delay k-j-delay) ) )
                (do
                    (?( (|| (! i-j-delay)
                            (< i-k-j-delay i-j-delay) )
                        (:= ([]a delay i j) i-k-j-delay)
                        (:= ([]a path i j) ([]a path  i k) ) )
                      ( (== i-k-j-delay i-j-delay)
                        (:= ([]a path i j)
                            (unionq &&& ([]a path i k))))))))))))

        () ) )


(defun me:me:delay ( me1 me2 )
    (?( (! me1)
        0)
      ( (! me2)
        0)
      ( t
        ([]a *sp.me:me:delay* (me:number me1) (me:number me2) ) ) ) )


(defun me:me:first-on-path ( me1 me2 )
    (?( (! me1)
        () )
      ( (! me2)
        () )
      ( t
        ([]a *sp.me:me:path* (me:number me1) (me:number me2) ) ) ) )


(defun shortest-path.print ()

(let ( (size (me-universe:size) ) )

    (loop (incr first-j from 0 to (- size 1) by 13)
          (bind last-j (min (+ first-j 12) (- size 1) ) )
    (do
        (msg 0 t)
        (let ( (names
                   (loop (incr j from first-j to last-j) (save
                       (sp.name:split (me:name (me-universe:number:me j) )
                                      t) ) ) ) )
            (msg 0 9)
```

```
            (loop (for (top-name bottom-name) in names) (do
                (msg "|" (jc top-name -4) ) ) )
            (msg 0 9)
            (loop (for (top-name bottom-name) in names) (do
                (msg "|" (jc bottom-name -4) ) ) ) )

        (msg 0)
        (loop (incr i from 0 to (- size 1) )
              (bind me-i (me-universe:number:me i)
                    (left-name right-name) (sp.name:split (me:name me-i) ()))
        (do
            (if (eqstr "" right-name) (then
                (msg (c left-name) (t 11) ) )
            (else
                (msg (c left-name) "//" (c right-name) (t 11) ) ) )

            (loop (incr j from first-j to last-j)
                  (bind me-j (me-universe:number:me j)
                        val  (me:me:delay me-i me-j) )
            (do
                (if val (then
                    (if (inump val) (then
                        (msg (j val -4)  " ") )
                    (else
                        (msg (f val 4 2) " ") ) ) )
                (else
                    (msg (jc "_" -5) ) ) ) ) )

            (msg t) ) ) ) )

    () ) )


;***=========================================================================
;***
;*** (SP.NAME:SPLIT NAME SPLIT-<=8?)
;***
;*** Auxiliary function used by LS.DELAY-TABLE:PRINT to produce pretty
;*** truncated labels for the margins.
;***
;***=========================================================================

(defun sp.name:split ( name split-<=8? )
    (?( (<= (stlength name) 4)
        `(,name "") )

      ( (<= (stlength name) 8)
        (if split-<=8?
            `(,(substring name 1 4)
              ,(substring name 5) )
            `(,name "") ) )

      ( t
        `(,(substring name 1 4)
          ,(substring name -4) ) ) ) )
```

```
;***-----------------------------------------------------------------********
;*** Copyright (C) 1983 John R. Ellis.  This file may be used for      ********
;*** educational and research purposes only, by the faculty, students, ********
;*** and staff of Yale University only.                                ********
;***-----------------------------------------------------------------********
;===========================================================================
;
; ELI MACHINE-MODEL
;
; A simple-minded, unrealistic ELI with 4 memory clusters, 4 floating
; clusters, almost all parameters hardwired.
;
;===========================================================================

(alias '^^ 'atomconcat)

(defun em.make-memory-cluster ( i n )
    (let ( (mi   (atomconcat 'm i) )
           (mi-1 (atomconcat 'm (mod (- i 1) n) ) )
           (mi+1 (atomconcat 'm (mod (+ i 1) n) ) )
           (fi   (atomconcat 'f i) )
           (fi-1 (atomconcat 'f (mod (- i 1) n) ) ) )
        `(
            (def-me (name                ,(^^ mi 'r) )
                    (type                register-bank)
                    (size                32)
                    (inputs              (,(^^ mi   '+)
                                          ,(^^ mi   'm)
                                          ,(^^ mi   'c)
                                          ,(^^ mi   'r)
                                          ,(^^ mi-1 'r)
                                          ,(^^ mi+1 'r)
                                          ,(^^ fi-1 'r)
                                          ,(^^ fi   'r) ) )
                    (read-ports      5)
                    (write-ports     4)
                    (read-resources  ( (,(^^ mi 'r-read) ) ) )
                    (write-resources ( (,(^^ mi 'r-write) ) ) ) )

            (def-resource-class ,(^^ mi 'r-read)  5)
            (def-resource-class ,(^^ mi 'r-write) 4)

            (def-me (name      ,(^^ mi '+) )
                    (type      functional-unit)
                    (delay     0)
                    (inputs    (,(^^ mi 'r)
                                ,(^^ mi 'c) ) )
                    (resources ( (,(^^ mi '+) ) ) )
                    (operators (if-ieq if-ine if-igt if-ige if-ile if-ilt
                                if-ie0mod if-true if-false iabs iadd iand
                                idiv ieq iexp ie0mod ige igt ile ilt imax
                                imin imul ine ineg inot ior isub bitrev) ) )

            (def-resource-class ,(^^ mi '+) 1)

            (def-me (name      ,(^^ mi 'm) )
                    (type      functional-unit)
                    (delay     2)
                    (inputs    (,(^^ mi 'r) ) )
                    (resources ( (,(^^ mi 'm) ) ) )
                    (operators (vbase ivload fvload ipload fpload ivstore
                                fvstore ipstore fpstore) ) )
```

```
            (def-resource-class ,(^^ mi 'm) 1)

            (def-me (name      ,(^^ mi 'c) )
                    (type      constant-generator)
                    (resources ( (,(^^ mi 'c) ) ) )
                    (constraint-function
                               (lambda ( constant )
                                     (|| (consp constant)
                                         (inump constant) ) ) ) ) )

            (def-resource-class ,(^^ mi 'c) 1)

            (def-bus ,(^^ mi-1 'r) ,(^^ mi 'r) 1)
            (def-bus ,(^^ mi+1 'r) ,(^^ mi 'r) 1)
            (def-bus ,(^^ fi-1 'r) ,(^^ mi 'r) 1)
            (def-bus ,(^^ fi   'r) ,(^^ mi 'r) 1)
            ) ) )


(defun em.make-floating-cluster ( i n )
    (let ( (mi   (atomconcat 'm i) )
           (mi+1 (atomconcat 'm (mod (+ i 1) n) ) )
           (fi   (atomconcat 'f i) ) )
        `(
            (def-me (name      ,(^^ fi 'r) )
                    (type      register-bank)
                    (size      32)
                    (inputs    (,(^^ fi   '+)
                                ,(^^ fi   '=)
                                ,(^^ fi   '*)
                                ,(^^ fi   'r)
                                ,(^^ mi   'r)
                                ,(^^ mi+1 'r) ) )
                    (read-ports      4)
                    (write-ports     3)
                    (read-resources  ( (,(^^ fi 'r-read) ) ) )
                    (write-resources ( (,(^^ fi 'r-write) ) ) ) )

            (def-resource-class ,(^^ fi 'r-read)  4)
            (def-resource-class ,(^^ fi 'r-write) 3)

            (def-me (name      ,(^^ fi '+) )
                    (type      functional-unit)
                    (delay     1)
                    (inputs    (,(^^ fi 'r) ) )
                    (resources ( (,(^^ fi '+) ) ) )
                    (operators (fabs fadd feq fge fgt fix fle float flt
                                fmax fmin fne fneg fsub) ) )

            (def-me (name      ,(^^ fi '=) )
                    (type      functional-unit)
                    (delay     0)
                    (inputs    (,(^^ fi 'r) ) )
                    (resources ( (,(^^ fi '+) ) ) )
                    (operators (if-feq if-fne if-fgt if-fge if-fle if-flt) ) )

            (def-resource-class ,(^^ fi '+) 1)

            (def-me (name      ,(^^ fi '*) )
                    (type      functional-unit)
                    (delay     2)
                    (inputs    (,(^^ fi 'r) ) )
                    (resources ( (,(^^ fi '*) ) ) )
```

```
                    (operators (fmul fdiv fexp cos sin sqrt tan) ) )

            (def-resource-class ,(^^ fi '*) 1)

            (def-bus ,(^^ mi   'r) ,(^^ fi 'r) 1)
            (def-bus ,(^^ mi+1 'r) ,(^^ fi 'r) 1)
            ) ) )


(eval
`(def-machine-model

    .,(loop (incr i from 0 to 8) (splice
        (em.make-memory-cluster i 4) ) )

    .,(loop (incr i from 0 to 8) (splice
        (em.make-floating-cluster i 4) ) )
    ) )
```

```
;***------------------------------------------------------------------********
;*** Copyright (C) 1983 John R. Ellis.  This file may be used for     ********
;*** educational and research purposes only, by the faculty, students, ********
;*** and staff of Yale University only.                               ********
;***------------------------------------------------------------------********
;==============================================================================
;
; MACHINE MODEL SIMULATOR
;
; This module implements the program simulator for the list-scheduler machine
; model.
;
; A program is a list of instructions.  An instruction is a list of
; operations.  An operation has the form:
;
;     (me operator . operands)
;
; ME is the name of some machine element, OPERATOR is the name of the
; operator to be executed, and OPERANDS are the operand specifiers.
;
; An operand has one of two forms:
;
;     (register-bank-me register name)    to access a register;
;
;     (constant-generator-me name)        to generate a constant;
;
; NAME is the name of the value in the register or being generated.  For
; registers, it is just a hint about what the symbolic contents of the
; register should be (the simulator remembers and checks those hints).
; For constant generators, NAME is the value that should be generated.
;
; A value name can be:
;
;     a simple variable, e.g. X
;
;     a number, e.g. 3.0
;
;     a VBASE expression, e.g. (VBASE V)
;
;     an address expression of the form (ADDRESS x) where "x" is any of the
;     above name forms.
;
; A sample operation:
;
;     (MO+ IADD (MOR 2 X) (MOR 4 (VBASE V) ) (MOC 1) )
;
; A continution operation (a "pusher") for a multi-cycle pipeline is
; represented as:
;
;     (F3+)
;
; A pseudo-operation doesn't a () ME:
;
;     (() LABEL L33)
;     (NIL DCL %F%I FLOAT 256 ((1 256)) NIL)
;------------------------------------------------------------------------------
;
; (SIMULATE PROGRAM ACTUALS)
;     Runs the simulator on PROGRAM with actual parameters ACTUALS (specified
;     the same as for the NADDR interpreter).
;
; (SIM.PRINT-PROGRAM-VARIABLES VAR-SPECS LOOK-IN-REGISTERS?)
;     VAR-SPECS is a list of program variables; prints out each variable
```

```
;     in turn.  Each spec in VAR-SPECS is either an atom (for a vector
;     or scalar) or a triple (VAR REAL IMAG) where VAR is the tiny lisp
;     name of a complex variable and REAL and IMAG are the NADDR variables
;     used to implement the real and imaginary parts.  If
;     LOOK-IN-REGISTERS?  is is true, then the contents of any registers
;     containing the variables are printed out as well; otherwise only
;     memory is examined.
;
; *SIM.OPERATION-COUNT*
; *SIM.INSTRUCTION-COUNT*
;     Total number of operations and instructions executed, excluding
;     pseudo-ops and instructions consisting only of pseudo-ops.
;
; (SVAR 'VAR1 'VAR2 ...)
;     Interactive syntax for printing out variables, equivalent to:
;
;             (SIM.PRINT-PROGRAM-VARIABLES '(VAR1 VAR2 ...) T)
;
;     which will look both in memory and in registers for the variables.
;
;
; ##R ME-NAME REGISTER
;     Interactive syntax for getting the contents of a register.
;
;==============================================================================

(eval-when (compile load)
    (include list-scheduler:declarations) )
(eval-when (compile)
    (build '(utilities:options) ) )


(def-option *sim.trace-instructions?* () list-scheduler: "
If non-(), then instructions will be printed out as they are executed.
If T, tracing begins with the first cycle.  If some number, then tracing
begins with that cycle.
")

(def-option *sim.step-instructions?* () list-scheduler: "
This is just like *SIM.TRACE-INSTRUCTIONS?*, except that a break-point
is invoked after each instruction traced.
")

(def-option *sim.trace-values?* () list-scheduler: "
If non-(), then values being stored in registers and memory are printed
out after the cycle in which the store actually occurs (i.e. at the
end of pipe execution.
")

(def-option *sim.break-label* () list-scheduler: "
If non-(), then the simulator will stop with a breakpoint if a label of
of this name is reached.
")



(declare (special
    *sim.pc*              ;*** These globals are mainly for debugging --
    *sim.running?*        ;*** when a breakpoint occurs, you can look at
    *sim.instr*           ;*** the current cycle, instruction, and
    *sim.oper*            ;*** operation.
    *sim.cycle*
```

```
    *sim.start-pc*          ;*** The pc to start execution at (where the
                            ;*** DEF-BLOCK is).

    *sim.significant-instruction?*
                                ;*** True if the current instruction
                                ;*** contained a non-pseudo-op.
    *sim.operation-count*       ;*** # operations initiated this cycle.
    *sim.copy-count*            ;*** # COPYs executed this cycle.

    *sim.def-block*
    ) )


(defun simulate ( program actuals )
    (registers.initialize)

    (sim.initialize-memory)
    (sim.initialize-variables)
    (sim.initialize-labels)
    (sim.initialize-pipelines)

    (:= *sim.def-block* () )
    (sim.program:declare program)

    (if *sim.def-block* (then
        (sim.formals:actuals:bind (oper:part *sim.def-block* 'in-variables)
                                  actuals) ) )

    (sim.program:execute *sim.start-pc*)

    (sim.print-program-variables (oper:part *sim.def-block* 'out-variables)
                                 () )
    () )


(def-inline sim.trace? (function ( trace-var )
    (&& trace-var
        (|| (! (inump trace-var) )
            (<= trace-var *sim.cycle*) ) ) ) )


;***=============================================================================
;***
;*** MEMORY
;***
;*** The simulator memory is represented as a vector, of course, from
;*** which space for variables, vectors, and constants is allocated.
;*** Memory is initialized to all zeroes, so that we can index past the
;*** end of an array.
;***
;*** (SIM.INITIALIZE-MEMORY)
;***     Initializes the memory to a default size.
;***
;*** (SIM.MEMORY:[] ADDR)
;***     Returns the contents of the ADDRth word.
;***
;*** (SIM.MEMORY:ALLOCATE LENGTH)
;***     Allocates a chunk of LENGTH words from memory, returning the
;***     address of the first word.  Memory is grown if its current size
;***     is too small.
;***
;*** (SIM.ADDRESS:COERCE ADDR)
;***     Coerces a memory address to be an integer between 0 and the size
```

```
;***     of memory - 1.  We do this because it's perfectly ok for compacted
;***     programs to generate bogus indices when they're doing nested
;***     vector references (A (L I) J) (as in SOLVE).  It would be better
;***     to detect vector-out-of-bounds and return 0, but we don't yet
;***     have the hooks in generated code to do that easily (no vector
;***     names).
;***
;***=============================================================================

(declare (special
    *sim.memory*
    *sim.next-free-address*
    ) )


(defun sim.initialize-memory ()
    (:= *sim.memory* (makevector 1000) )
    (vector:initialize *sim.memory* 0)
    (:= *sim.next-free-address* 0)
    () )


(defmacro sim.memory:[] ( addr )
    '([] *sim.memory* ,addr) )


(defun sim.memory:allocate ( length )
    (let ( (base        *sim.next-free-address*)
           (max-length (vectorlength *sim.memory*) ) )

        (:= *sim.next-free-address* (+ &&& length) )
        (if (> *sim.next-free-address* max-length) (then
            (:= *sim.memory* (vector:copy &&& (* 2 max-length) ) )
            (loop (incr i from max-length to (- (vectorlength *sim.memory*) 1))
            (do
                (:= ([] *sim.memory* i) 0) ) ) ) )
        base) )


(defun sim.address:coerce ( addr )
    (if (&& (inump addr)
            (>= addr 0)
            (<  addr *sim.next-free-address*) )
    (then
        addr)
    (else
        0) ) )


;***=============================================================================
;***
;*** VARIABLES
;***
;*** Information about variable names and constants is kept in a hash
;*** table mapping the name or constant onto a record of type SIM.VD which
;*** describes the variable or constant.  The only variables actually
;*** stored in memory are input and output variables and vectors.
;*** Constants are stored in memory only if they are contained in an
;*** (ADDRESS constant) expression.
;***
;*** (SIM.INITIALIZE-VARIABLES)
;***     Forgets all previous variable information.
;***
```

```
;*** (SIM.VARIABLE:DECLARE NAME LENGTH DIMENSIONS INITIAL-VALUES )
;***     Declares NAME (a constant or variable), where LENGTH, DIMENSIONS,
;***     and INITIAL-VALUES are as specified on a DCL pseudo-op.  Space
;***     is allocated in memory for the variable or constant and
;***     initialized to the initial values.
;***
;*** (SIM.VARIABLE:INITIALIZE VARIABLE INITIAL-LIST )
;***     VARIABLE should already have been declared.  Initializes its
;***     memory with the values taken from INITIAL-LIST;  if the list is
;***     () the variable is initialized to all zeroes.
;***
;*** (SIM.VARIABLE:VD         VARIABLE)
;*** (SIM.VARIABLE:BASE       VARIABLE)
;*** (SIM.VARIABLE:LENGTH     VARIABLE)
;*** (SIM.VARIABLE:DIMENSIONS VARIABLE)
;*** (SIM.VARIABLE:RANK       VARIABLE)
;***     These functions return the declared information about a variable.
;***     An error is raised if the variable isn't declared.
;***
;***============================================================================

(declare (special
    *sim.variable:vd*   ;*** Hash table mapping names to SIM.VD records.
    ) )

(def-struct sim.vd      ;*** A descriptor for a variable.
    name                ;*** Its name.
    base                ;*** The address of its first word.
    length              ;*** Its length.
    dimensions          ;*** Its dimensions list, () if a scalar.
    rank                ;*** The number of dimensions.
    )


(defun sim.initialize-variables ()
    (:= *sim.variable:vd* (hash-table:create 'eql () () ) )
    () )


(defun sim.variable:vd ( variable )
    (|| ([]h *sim.variable:vd* variable)
        (error (list variable "SIMULATOR: Variable isn't declared.") ) ) )


(defun sim.variable:base ( variable )
    (sim.vd:base (sim.variable:vd variable) ) )

(defun sim.variable:length ( variable )
    (sim.vd:length (sim.variable:vd variable) ) )

(defun sim.variable:dimensions ( variable )
    (sim.vd:dimensions (sim.variable:vd variable) ) )

(defun sim.variable:rank ( variable )
    (sim.vd:rank (sim.variable:vd variable) ) )


(defun sim.variable:declare ( name length dimensions initial-values )
    (if-let ( (nd ([]h *sim.variable:vd* name) ) ) (then
        (assert (&& (== length     (sim.vd:length     nd) )
                    (== dimensions (sim.vd:dimensions nd) ) )
            (h name) " " (h length) " " (h dimensions) " " t
            (h nd) t
```

```
            "SIMULATOR:  Name previously declared with other attributes.") )
    (else
        (let ( (nd (sim.vd:new
                        name       name
                        length     length
                        dimensions dimensions
                        rank       (length dimensions)
                        base       (sim.memory:allocate length) ) ) )
            (:= ([]h *sim.variable:vd* name) nd)
            (sim.variable:initialize name initial-values) ) ) )
    () )


(defun sim.variable:initialize ( variable initial-list )
    (?( (! initial-list)
        (:= initial-list '(0) ) )
      ( (! (consp initial-list) )
        (:= initial-list (list initial-list) ) ) )

    (loop (incr i from (sim.variable:base variable)
                  to   (+ -1 (+ (sim.variable:base   variable)
                                (sim.variable:length variable) ) ) )
          (initial value               0
                   rest-initial-list initial-list)
    (do
        (if (! rest-initial-list)
            (:= rest-initial-list initial-list) )
        (:= value (pop rest-initial-list) )
        (:= (sim.memory:[] i) value) ) )

    () )


;***============================================================================
;***
;*** LABELS
;***
;*** Program labels are represented as Lisp symbols.  A hash table maps the
;*** symbols onto the corresponding program locations.  A program location
;*** is just some tail of the list of instructions representing the program.
;***
;*** (SIM.LABELS:INITIALIZE)
;***     Forgets all information about labels.
;***
;*** (SIM.LABEL:DECLARE LABEL PC)
;***     Declares LABEL to be at program location PC.
;***
;*** (SIM.LABEL:PC LABEL)
;***     Returns the program location of a label.
;***
;***============================================================================


(declare (special
    *sim.label:pc*
    ) )

(defun sim.initialize-labels ()
    (:= *sim.label:pc*  (hash-table:create () () () ) )
    () )


(defun sim.label:declare ( label pc )
```

```
    (assert (! ([]h *sim.label:pc* label) )
        (h label) t "SIMULATOR: Label already declared.")
    (:= ([]h *sim.label:pc* label) pc)
    () )


(defun sim.label:pc ( label )
    (|| ([]h *sim.label:pc* label)
        (error (list label "SIMULATOR: Label not declared.") ) ) )


;***==========================================================================
;***
;*** OPERANDS
;***
;*** See above for how operands are represented in the program.
;***
;*** (SIM.OPERAND:DECLARE OPERAND)
;***     Declares any constant mentioned in the OPERAND, allocating and
;***     initializing a memory location to contain it.  Only constants
;***     within (ADDRESS ...)  expressions in the operand need be so
;***     allocated.
;***
;*** (SIM.OPERAND:VALUE OPERAND)
;***     Returns the value of an operand, which is the contents of the
;***     specified register or the constant generated.  For register
;***     operands, the name mentioned in the operand is checked against
;***     the symbolic name contained in the register; if they're not the
;***     same, an error is raised.
;***
;***==========================================================================

(defun sim.operand:declare ( operand )
    (let ( ( (cg name) operand) )
        (if (&& (== 2 (length operand) )
                (== 'constant-generator (me:type (name:me cg) ) )
                (consp name)
                (== 'address (car name) ) )
        (then
            (let ( ( (address item) name) )
                (?( (litatom item)
                    (sim.variable:declare item 1 () () ) )
                  ( (numberp item)
                    (sim.variable:declare item 1 () item) )
                  ( (&& (consp item)
                        (== 'vbase (car item) ) )
                    (let ( (vd (sim.variable:vd (cadr item) ) ) )
                        (assert (< 0 (sim.vd:rank vd) )
                            (h item) t "SIMULATOR: Invalid VBASE name.")
                        (sim.variable:declare item 1 () (sim.vd:base vd))))
                  ( t
                    (error (list name "SIMULATOR: Invalid operand.")))))))
        () ) )


(defun sim.operand:value ( operand )
    (let*( ( (me-name . args) operand)
           ( me                 (name:me me-name) ) )

        (caseq (me:type me)
            (constant-generator
                (let ( ( (constant) args) )
                    (if (consp constant) (then
```

```
                        (caseq (car constant)
                            ( (vbase address)
                                (sim.variable:base (cadr constant) ) )
                            (t
                                (error (list operand "Case error.") ) ) ) ) )
                    (else
                        constant) ) ) )
            (register-bank
                (let ( ( (register name) args) )
                    (if (!= name ([] (me:value-names me) register) ) (then
                        (msg (h operand) t
                            "SIMULATOR: Name of value in register is different "
                            "from name of operand.")
                        (break-point simulator) ) )
                    ([] (me:values me) register) ) )
            (t
                (error (list operand "SIMULATOR: Invalid operand."))))))


;***==========================================================================
;***
;*** PIPELINES
;***
;*** The results from functional unit pipelines are not written into their
;*** destinations until after the cycle from which they emerge from the
;*** pipe.  To implement this behavior, we keep two priority queues, one
;*** for registers and one for memory.  Entries in each queue are
;*** destination/value pairs, keyed by the cycle in which the value is
;*** to be stored into the destination.  When an operation is executed,
;*** it's result is entered into the appropriate priority queue with the
;*** cycle it should be stored.  At the end of each instruction cycle,
;*** the simulator stores away all the results queued for that cycle.
;***
;*** (SIM.INITIALIZE-PIPELINES)
;***     Initializes the queues representing the pipelines.
;***
;*** (SIM.DELAY:REGISTER-OPERAND:VALUE:STORE DELAY OPERAND VALUE)
;***     Remembers the fact that VALUE should be stored into the register
;***     destination described by OPERAND (see above) at the end of the
;***     cycle which is DELAY cycles after the current cycle.
;***
;*** (SIM.DELAY:ADDRESS:VALUE:STORE DELAY ADDRESS VALUE)
;***     Remembers the fact that VALUE should be stored in memory location
;***     ADDRESS at the end of the cycle which is DELAY cycles after
;***     the current cycle.
;***
;*** (SIM.STORE-SAVED-RESULTS)
;***     Removes all entries from the two priority queues for the current
;***     cycle and stores their values in the given destinations.
;***
;***==========================================================================

(declare (special
    *sim.saved-register-cycles*
    *sim.saved-register-values*
    *sim.saved-register-operands*
    *sim.next-saved-register-slot*

    *sim.saved-memory-cycles*
    *sim.saved-memory-values*
    *sim.saved-memory-addresses*
    *sim.next-saved-memory-slot*
    ) )
```

```
(defun sim.initialize-pipelines ()
    (:= *sim.saved-register-cycles*    (makevector 100) )
    (:= *sim.saved-register-values*    (makevector 100) )
    (:= *sim.saved-register-operands*  (makevector 100) )
    (:= *sim.next-saved-register-slot* 0)

    (:= *sim.saved-memory-cycles*      (makevector 100) )
    (:= *sim.saved-memory-values*      (makevector 100) )
    (:= *sim.saved-memory-addresses*   (makevector 100) )
    (:= *sim.next-saved-memory-slot*   0)
    () )


(defun sim.delay:register-operand:value:store ( delay operand value )
    (let ( (cycle (+ *sim.cycle* delay) ) )
        (:= ([] *sim.saved-register-cycles*   *sim.next-saved-register-slot*)
            cycle)
        (:= ([] *sim.saved-register-operands* *sim.next-saved-register-slot*)
            operand)
        (:= ([] *sim.saved-register-values*   *sim.next-saved-register-slot*)
            value)
        (:= *sim.next-saved-register-slot* (+ &&& 1) )
        () ) )


(defun sim.delay:address:value:store ( delay address value )
    (let ( (cycle (+ *sim.cycle* delay) ) )
        (:= ([] *sim.saved-memory-cycles*    *sim.next-saved-memory-slot*)
            cycle)
        (:= ([] *sim.saved-memory-addresses* *sim.next-saved-memory-slot*)
            address)
        (:= ([] *sim.saved-memory-values*    *sim.next-saved-memory-slot*)
            value)
        (:= *sim.next-saved-memory-slot* (+ &&& 1) )
        () ) )


(defun sim.store-saved-results ()
    (loop (initial next-slot 0)
          (incr i from 0 to (- *sim.next-saved-register-slot* 1) )
          (bind cycle   ([] *sim.saved-register-cycles*   i)
                operand ([] *sim.saved-register-operands* i)
                value   ([] *sim.saved-register-values*   i) )
    (do
        (if (== cycle *sim.cycle*) (then
            (let*( ( (me-name register name) operand)
                   ( me                      (name:me me-name) ) )
                (:= ([] (me:values      me) register) value)
                (:= ([] (me:value-names me) register) name)

                (if (sim.trace? *sim.trace-values?*) (then
                    (msg 0 (j operand -20) " = " value t) ) ) ) )

        (else
            (:= ([] *sim.saved-register-cycles*   next-slot) cycle)
            (:= ([] *sim.saved-register-operands* next-slot) operand)
            (:= ([] *sim.saved-register-values*   next-slot) value)
            (:= next-slot (+ &&& 1) ) ) ) )
    (result
        (:= *sim.next-saved-register-slot* next-slot) ) )
```

```
    (loop (initial next-slot 0)
          (incr i from 0 to (- *sim.next-saved-memory-slot* 1) )
          (bind cycle   ([] *sim.saved-memory-cycles*    i)
                address ([] *sim.saved-memory-addresses* i)
                value   ([] *sim.saved-memory-values*    i) )
    (do
        (if (== cycle *sim.cycle*) (then
            (:= (sim.memory:[] address) value)

            (if (sim.trace? *sim.trace-values?*) (then
                (msg 0 (j address -20) " = " value t) ) ) )

        (else
            (:= ([] *sim.saved-memory-cycles*    next-slot) cycle)
            (:= ([] *sim.saved-memory-addresses* next-slot) address)
            (:= ([] *sim.saved-memory-values*    next-slot) value)
            (:= next-slot (+ &&& 1) ) ) ) )
    (result
        (:= *sim.next-saved-memory-slot* next-slot) ) )

    () )


;***=========================================================================
;***
;*** (SIM.PROGRAM:DECLARE PROGRAM)
;***
;*** Scans all the operations in PROGRAM, declaring all the operands, allocating
;*** and initializing memory for constants, input and output variables, and
;*** vectors.  Labels are found and declared, and the DEF-BLOCK is put
;*** in *SIM.DEF-BLOCK*.
;***
;***=========================================================================

(defun sim.program:declare ( program )
    (:= *sim.largest-trace* 1)
    (:= *sim.start-pc*      program)

    (loop (initial *sim.pc* program)
          (while *sim.pc*)
          (bind instr (car *sim.pc*) )
    (do
        (sim.instr:declare-pass-1 instr) )
    (next *sim.pc* (cdr *sim.pc*) ) )


    (loop (initial *sim.pc* program)
          (while *sim.pc*)
          (bind instr (car *sim.pc*) )
    (do
        (sim.instr:declare-pass-2 instr) )
    (next *sim.pc* (cdr *sim.pc*) ) )

    () )


(defun sim.instr:declare-pass-1 ( instr )
    (loop (for oper in instr) (do
        (sim.oper:declare-pass-1 oper) ) )
    () )


(defun sim.oper:declare-pass-1 ( oper )
```

```
    (let*( ( (me-name . naddr-oper) oper)
           ( (operator . operands)  naddr-oper) )
        (?( (! me-name)
            (caseq operator
                (def-block
                    (:= *sim.def-block* '(,operator ,.operands) )
                    (:= *sim.start-pc*  *sim.pc*) )
                (label
                    (sim.label:declare (car operands) *sim.pc*) )
                (dcl
                    (sim.variable:declare
                        (oper:part naddr-oper 'variable)
                        (oper:part naddr-oper 'length)
                        (oper:part naddr-oper 'dimensions)
                        (oper:part naddr-oper 'initial-list) ) )
                (trace
                    (let ( ( (number . () ) operands) )
                        (:= *sim.largest-trace* (max &&& number)))))))
        () ) )


(defun sim.instr:declare-pass-2 ( instr )
    (loop (for oper in instr) (do
        (sim.oper:declare-pass-2 oper) ) )
    () )


(defun sim.oper:declare-pass-2 ( oper )
    (let*( ( (me-name . naddr-oper) oper)
           ( (operator . operands)  naddr-oper) )
        (?( (! me-name)
            () )

          ( (! naddr-oper)
            () )

          ( (== 'copy operator)
            (sim.operand:declare (cadr operands) ) )

          ( t
            (caseq (operator:group operator)
                ( (one-in-one-out two-in-one-out vload)
                    (loop (for operand in (cdr operands) ) (do
                        (sim.operand:declare operand) ) ) )
                (vstore
                    (loop (for operand in operands) (do
                        (sim.operand:declare operand) ) ) )
                (if-compare
                    (sim.operand:declare (car  operands) )
                    (sim.operand:declare (cadr operands) ) )
                (if-boolean
                    (sim.operand:declare (car  operands) ) ) ) ) )

        () ) )

;***===================================================================
;***
;*** (SIM.PROGRAM:EXECUTE START-PC)
;***
;*** This is the main execute loop, executing instructions until we run
;*** off the end of the program or we hit a STOP.
;***
```

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

(defun sim.program:execute ( start-pc )
    (sim.initialize-execution-records)

    (loop (incr *sim.cycle* from 0)
          (initial *sim.running?* t
                   pc             start-pc)
          (while *sim.running?*)
          (while pc)
    (do
        (:= *sim.pc* () )
        (sim.instr:execute (car pc) )
        (:= pc (|| *sim.pc* (cdr pc) ) ) ) )
    () )


;***=========================================================================
;***
;*** (SIM.INSTR:EXECUTE *SIM.INSTR*)
;***
;*** Executes a single instruction.
;***
;***=========================================================================

(defun sim.instr:execute ( *sim.instr* )
    (:= *sim.significant-instruction?* () )
    (:= *sim.operation-count*          0)
    (:= *sim.copy-count*               0)

    (if (|| (sim.trace? *sim.trace-instructions?*)
            (sim.trace? *sim.step-instructions?*) )
    (then
        (msg 0 (j *sim.cycle* 4) ": " (h *sim.instr*) t)
        (if (sim.trace? *sim.step-instructions?*) (then
            (break-point single-step) ) ) ) )

    (loop (for oper in *sim.instr*) (do
        (sim.oper:execute oper) ) )
    (sim.store-saved-results)

    (sim.record-instruction-execution)
    () )


;***=========================================================================
;***
;*** (SIM.OPER:EXECUTE *SIM.OPER*)
;***
;*** Executes a single operation by dispatching according to type:
;*** pseudo-operation, register-bank operation (a copy), or functional-unit
;*** operation.
;***
;***=========================================================================

(defun sim.oper:execute ( *sim.oper* )
    (let*( ( (me-name . () ) *sim.oper*) )
        (if (! me-name) (then
            (sim.pseudo-op:execute *sim.oper*) )
        (else
            (let ( (me (name:me me-name) ) )
                (caseq (me:type me)
                    (register-bank
```

```
                        (sim.register-bank-me:oper:execute me *sim.oper*) )
                    (functional-unit
                        (sim.functional-unit-me:oper:execute me *sim.oper*) )
                    (t
                        (error (list *sim.oper*
                            "SIMULATOR: Invalid ME in machine operation.")))))
            ) )
        () ) )


;***==========================================================================
;***
;*** (SIM.PSEUDO-OP:EXECUTE OPER)
;***
;*** Executes a pseudo-operation.
;***
;***==========================================================================

(defun sim.pseudo-op:execute ( oper )
    (let ( ( (() operator . operands) oper) )
        (caseq operator
            (label
                (if (== (car operands) *sim.break-label*) (then
                    (break-point simulator-label-break) ) ) )
            (trace
                (let ( ( (number type . () ) operands) )
                    (:= *sim.current-trace*      number)
                    (:= *sim.current-trace-type* type) ) )
            (goto
                (if (! *sim.pc*) (then
                    (:= *sim.pc* (sim.label:pc (car operands) ) ) ) ) )
            (stop
                (:= *sim.running?* () ) )
            (esc
                (eval '(progn .,operands) ) ) )
        () ) )


;***==========================================================================
;***
;*** (SIM.REGISTER-BANK-ME:OPER:EXECUTE ME OPER)
;***
;*** Executes a register bank operation for register bank ME.  The only
;*** defined operation for register banks is COPY.
;***
;***==========================================================================

(defun sim.register-bank-me:oper:execute ( me (() copy dest source) )
    (assert (== 'copy copy)
        (h *sim.oper* "SIMULATOR: Unknown register-bank operator.") )
    (sim.delay:register-operand:value:store
        0
        dest
        (sim.operand:value source) )

    (:= *sim.copy-count*                (+ &&& 1) )
    (:= *sim.significant-instruction?* t)
    () )


;***==========================================================================
;***
;*** (SIM.FUNCTIONAL-UNIT-ME:OPER:EXECUTE ME OPER)
```

```
;***
;*** Executes a functional unit operation for functional unit ME.  We
;*** cheat here by invoking the private interfaces to the NADDR interpreter
;*** to do most of our work.  This is the right approach, but unfortunately
;*** those interfaces are currently private and undocumented.  Gross.
;***
;***==========================================================================

(defun sim.functional-unit-me:oper:execute ( me (() operator . operands) )
    (:= *sim.significant-instruction?* t)

    (if (! operator) (then
        () )
    (else
        (:= *sim.operation-count* (+ &&& 1) )

        (let*( (naddr.operator (operator:naddr.operator operator) )
               (delay          (me:delay me) )
               (function       (naddr.operator:execute-function
                                    naddr.operator) ) )

            (caseq (naddr.operator:group naddr.operator)
                (one-in-one-out
                    (sim.delay:register-operand:value:store
                        delay
                        (car operands)
                        (funcall function
                                 (sim.operand:value (cadr operands)))))

                (two-in-one-out
                    (sim.delay:register-operand:value:store
                        delay
                        (car operands)
                        (funcall function
                                 (sim.operand:value (cadr  operands) )
                                 (sim.operand:value (caddr operands) ) ) ) )

                (vload
                    (caseq operator
                        ( (ivload fvload)
                            (let ( ( (dest vector index) operands) )
                                (sim.delay:register-operand:value:store
                                    delay
                                    dest
                                    (sim.memory:[]
                                        (sim.address:coerce
                                            (+ (sim.operand:value vector)
                                               (sim.operand:value index))))))

                        ( (ipload fpload)
                            (let ( ( (dest index) operands) )
                                (sim.delay:register-operand:value:store
                                    delay
                                    dest
                                    (sim.memory:[]
                                        (sim.address:coerce
                                            (sim.operand:value index))))))
                        (t
                            (error (list *sim.oper* "Case error.") ) ) ) )

                (vstore
                    (caseq operator
                        ( (ivstore fvstore)
```

```
                            (let ( ( (vector index source) operands) )
                                (sim.delay:address:value:store
                                    0   ;*** correct.
                                    (+ (sim.operand:value vector)
                                       (sim.operand:value index) )
                                    (sim.operand:value source) ) ) )

                        ( (ipstore fpstore)
                            (let ( ( (index source) operands) )
                                (sim.delay:address:value:store
                                    0   ;*** correct.
                                    (sim.operand:value index)
                                    (sim.operand:value source) ) ) )

                        (t
                            (error (list *sim.oper* "Case error.") ) ) ) )

                (if-compare
                    (if (! *sim.pc*) (then
                        (let*( ( (operand1 operand2 label1 label2) operands)
                               (label (funcall function
                                                (sim.operand:value operand1)
                                                (sim.operand:value operand2)
                                                label1
                                                label2) ) )
                             (if label (then
                                 (:= *sim.pc* (sim.label:pc label)))))))

                (if-boolean
                    (if (! *sim.pc*) (then
                        (let*( ( (operand1 label1 label2) operands)
                               (label (funcall function
                                                (sim.operand:value operand1)
                                                label1
                                                label2) ) )
                             (if label (then
                                 (:= *sim.pc* (sim.label:pc label)))))))

                (t
                    (error (list *sim.oper* "Case error.") ) ) ) ) ) )
    () )


;***============================================================================
;***
;*** (SIM.FORMALS:ACTUALS:BIND FORMALS ACTUALS)
;***
;*** Binds the actual parameters to the formal parameters of a program.
;*** FORMALS is the list of formal parameters as given in DEF-BLOCK.
;*** ACTUALS is the list of actual parameters as given to INTERPRET.
;***
;***============================================================================

(defun sim.formals:actuals:bind ( formals actuals )
    (assert (listp formals) )
    (assert (listp actuals) )

    (loop (for formal in formals)
          (for actual in actuals)
    (do
        (?      ;*** A non-complex scalar or array?
                ;***
```

```
          ( (litatom formal)
            (sim.variable:initialize formal actual) )

                ;*** A complex scalar?
                ;***
          ( (&& (consp formal)
                (== 3 (length formal) )
                (for-every (var in formal) (litatom var)  )
                (consp actual)
                (== 2 (length actual) )
                (numberp (car  actual) )
                (numberp (cadr actual) ) )
            (sim.variable:initialize (cadr  formal) (car  actual) )
            (sim.variable:initialize (caddr formal) (cadr actual) ) )

                ;*** A complex array?
                ;***
          ( (&& (consp formal)
                (== 3 (length formal) )
                (for-every (var in formal) (litatom var)  )
                (consp actual)
                (for-every (pair in actual)
                    (&& (consp pair)
                        (numberp (car  pair) )
                        (numberp (cadr pair) ) ) ) )
            (sim.variable:initialize (cadr formal)
                                      (loop (for (real imag) in actual) (save
                                         real) ) )
            (sim.variable:initialize (caddr formal)
                                      (loop (for (real imag) in actual) (save
                                         imag) ) ) )

          ( t
            (error (list formal actual
                         "SIMULATOR: Invalid formal/actual pair.") ) ) ) ) )
    () )


;***============================================================================
;***
;*** EXECUTION RECORDS
;***
;*** A SIM.ER records the execution statistics of a trace (or the entire
;*** program).
;***
;***====================;***
(def-struct sim.er      ;***
    (number           0) ;*** Number of the trace
                        ;***
    (entries          0) ;*** # times the trace was entered.
                        ;***
    (instructions     0) ;*** # instructions executed.
                        ;***
    (operations       0) ;*** # operations initiating a functional unit.
                        ;***
    (copies           0) ;*** # COPY operations.
                        ;***
    (ps-instructions 0) ;*** The fields keep the same counts for split- and
    (ps-operations   0) ;*** and join- partial schedules; these counts are
    (pj-instructions 0) ;*** included in the totals above.
    (pj-operations   0) ;***
    )                   ;***
;***====================;***
```

```
;***
;*** (SIM.INITIALIZE-EXECUTION-RECORDS)
;***     Initializes the statistics recording.
;***
;*** (SIM.RECORD-INSTRUCTION-EXECUTION)
;***     This is called after executing an instruction.  It just bumps the
;***     execution totals for the current trace, using the global variables
;***     *SIM.OPERATION-COUNT* and *SIM.COPY-COUNT*.
;***
;*** (SIMULATOR.PRINT-EXECUTION-STATISTICS)
;***     Dumps out all the execution statistics, including a per-trace profile.
;***
;***=========================================================================

(declare (special
    *sim.all-ers*                   ;*** All execution records.
    *sim.trace:er*                  ;*** Vector mapping trace numbers to
                                    ;*** execution records.
    *sim.largest-trace*             ;*** Largest trace number in the program.
    *sim.current-trace*             ;*** Current trace number.
    *sim.last-trace*                ;*** Previous trace number.
    *sim.current-trace-type*        ;*** SPLIT, JOIN, or ().

    ) )


(defun sim.initialize-execution-records ()
    (:= *sim.current-trace*      1)
    (:= *sim.last-trace*         () )
    (:= *sim.current-trace-type* () )

    (:= *sim.trace:er* (makevector (+ 1 *sim.largest-trace*) ) )

    (:= *sim.all-ers* () )
    (loop (incr i from 1 to *sim.largest-trace*) (do
        (:= ([] *sim.trace:er* i) (sim.er:new number i) )
        (push *sim.all-ers* ([] *sim.trace:er* i) ) ) )
    () )


(defun sim.record-instruction-execution ()
    (if *sim.significant-instruction?* (then
        (let ( (er ([] *sim.trace:er* *sim.current-trace*) ) )
            (:= (sim.er:instructions er) (+ &&& 1) )
            (:= (sim.er:operations   er) (+ &&& *sim.operation-count*) )
            (:= (sim.er:copies       er) (+ &&& *sim.copy-count*) )

            (if (!== *sim.current-trace* *sim.last-trace*) (then
                (:= (sim.er:entries er) (+ &&& 1) ) ) )
            (:= *sim.last-trace* *sim.current-trace*)

            (caseq *sim.current-trace-type*
                (split
                    (:= (sim.er:ps-instructions er)
                        (+ &&& 1) )
                    (:= (sim.er:ps-operations   er)
                        (+ &&& *sim.operation-count*) ) )
                (join
                    (:= (sim.er:pj-instructions er)
                        (+ &&& 1) )
                    (:= (sim.er:pj-operations   er)
                        (+ &&& *sim.operation-count*) ) )
                ( () )
```

```
                ( t
                  (error (list *sim.current-trace-type* "Case error."))))))
    () )


(defun simulator.print-execution-statistics ()

(let ( (sorted-ers (sort *sim.all-ers*
                         (f:l ( er1 er2 )
                            (> (sim.er:instructions er1)
                               (sim.er:instructions er2) ) ) ) )
      (t-er       (sim.er:new number "Tot") ) )

    (loop (for er in *sim.all-ers*) (do
        (:= (sim.er:instructions    t-er) (+ &&& (sim.er:instructions    er)))
        (:= (sim.er:operations      t-er) (+ &&& (sim.er:operations      er)))
        (:= (sim.er:copies          t-er) (+ &&& (sim.er:copies          er)))
        (:= (sim.er:entries         t-er) (+ &&& (sim.er:entries         er)))
        (:= (sim.er:ps-instructions t-er) (+ &&& (sim.er:ps-instructions er)))
        (:= (sim.er:ps-operations   t-er) (+ &&& (sim.er:ps-operations   er)))
        (:= (sim.er:pj-instructions t-er) (+ &&& (sim.er:pj-instructions er)))
        (:= (sim.er:pj-operations   t-er) (+ &&& (sim.er:pj-operations   er)))
        ) )

    (msg 0 t "EXECUTION PROFILE" t)
    (sim.er:print-heading)
    (msg (jc "-" 79 #/-) t)
    (sim.er:print t-er t-er)
    (msg (jc "-" 79 #/-) t)

    (loop (for er in sorted-ers)
          (incr i from 1)
          (when (> (sim.er:instructions er) 0) )
    (do
        (sim.er:print er t-er (if (== 0 (mod i 3) ) #/. #/ ) ) ) )
    (msg (jc "-" 79 #/-) t)

    () ) )


(defun sim.er:print-heading ()
    (msg (jc "Tr"       3)
         (jc "Instrs"   7)
         (jc "%"        4)
         (jc "Opers"    7)
         (jc "%"        4)
         (jc "Copies"   7)
         (jc "%"        4)
         (jc "PS-I"     6)
         (jc "%I"       3)
         (jc "PJ-I"     6)
         (jc "%I"       3)
         (jc "Entries"  9)
         (jc "I//E"     7)
         (jc "O//I"     5)
         (jc "C//I"     5)
         t)
    () )


(defun sim.er:print ( er t-er pc )

(msg (jc (sim.er:number er)                                              3 pc)
```

```
     (jc (sim.er:instructions er)                                         7 pc)
     (jc (%r (sim.er:instructions er) (sim.er:instructions t-er) )        4 pc)
     (jc (sim.er:operations er)                                           7 pc)
     (jc (%r (sim.er:operations er) (sim.er:operations t-er) )            4 pc)
     (jc (sim.er:copies er)                                               7 pc)
     (jc (%r (sim.er:copies er) (sim.er:copies t-er) )                    4 pc)
     (jc (sim.er:ps-instructions er)                                      6 pc)
     (jc (%r (sim.er:ps-instructions er) (sim.er:instructions t-er) ) 3 pc)
     (jc (sim.er:pj-instructions er)                                      6 pc)
     (jc (%r (sim.er:pj-instructions er) (sim.er:instructions t-er) ) 3 pc)
     (jc (sim.er:entries er)                                              8 pc)
     (f  (//f (sim.er:instructions er) (sim.er:entries er) )              7 1 pc)
     (f  (//f (sim.er:operations er) (sim.er:instructions er) )           5 1 pc)
     (f  (//f (sim.er:copies er)     (sim.er:instructions er) )           5 1 pc)
     t)
() )


(defun % ( x y )
    (if (= y 0)
        0
        (// (* x 100.0) y) ) )

(defun %r ( x y )
    (round (% x y) ) )

(defun //f ( x y )
    (if (= y 0)
        0
        (// (flonum x) y) ) )

;***============================================================================
;***
;*** VARIABLE PRINTING (see above).
;***
;***============================================================================

(defun sim.print-program-variables ( var-specs look-in-registers? )
    (loop (for var-spec in var-specs) (do
        (sim.var-spec:print var-spec look-in-registers?) ) )
    () )


(defmacro svar var-specs
    `(sim.print-program-variables ',var-specs t) )


(defun sim.var-spec:print ( var-spec look-in-registers? )
    (?( (&& look-in-registers?
            (|| (! ([]h *sim.variable:vd* var-spec) )
                (== 0 (sim.variable:rank var-spec) ) ) )
        (sim.var:find-registers var-spec) )

      ( (consp var-spec)
        (sim.var:real:imag:print
            (car var-spec) (cadr var-spec) (caddr var-spec) ) )

      ( t
        (sim.var:real:imag:print var-spec var-spec () ) ) )
    () )
```

```
(defun sim.var:find-registers ( var )
    (loop (for me in *ls.register-bank-mes*)
          (initial found-one? () )
    (do
        (loop (incr i from 0 to (- (me:size me) 1) )
              (when (= var ([] (me:value-names me) i) ) )
        (do
            (if (! found-one?) (then
                (msg 0 "Variable " var " ") ) )
            (msg (t 20) " = " ([] (me:values me) i) " "
                 (t 35) `(,(me:name me) ,i) t)
            (:= found-one? t) ) ) )
    (result
        (if (! found-one?) (then
            (msg 0 "Variable " var " not found.") ) ) ) )
    () )


(defun sim.var:real:imag:print ( var real imag )

(let ( (real-base (sim.variable:base real) )
       (imag-base (if imag (sim.variable:base imag) () ) ) )

    (if (! (sim.variable:dimensions real) ) (then
        (msg 0 "Variable " var " = "
             (sim.real:imag:offset:value real-base imag-base 0) t) )

    (else
            ;*** an array, print out each element of the array, labeled
            ;*** with its index.
            ;***
        (let ( (dimensions () ) )

                ;*** First make a list of triples of the form (LOWER UPPER
                ;*** SIZE) corresponding to the dimensions of the array.
                ;***
            (loop (for (lower upper)
                       in (reverse (sim.variable:dimensions real) ) )
                  (initial size  1)
            (do
                (push dimensions `(,lower ,upper ,size) )
                (:= size (* size (+ 1 (- upper lower) ) ) ) ) )

                ;*** Now print each element of the array, labeled with
                ;*** the index.
                ;***
            (msg "Variable " var " = " t)
            (loop (incr i from 0)
                  (bind    value        (sim.real:imag:offset:value
                                            real-base imag-base i)
                           last-i       (- i 1) )
                  (initial first-value  `(unique-garbage)
                           first-i      ()
                           final-i      (- (sim.variable:length real) 1) )
                  (when (|| (> i final-i)
                            (&& (!= value first-value) ) ) )
            (do
                (if first-i (then
                    (msg "   ")
                    (if (> last-i first-i) (then
                        (sim.index:print first-i var dimensions)
                        (msg " :") ) )
                    (sim.index:print last-i var dimensions)
```

```
                    (msg (t 18) " = " first-value t) ) )
                (if (> i final-i) (then
                    (return () ) ) )
                (:= first-i     i)
                (:= first-value value) ) ) ) ) )
    () ) ) )


(defun sim.index:print ( i var dimensions )
    (loop (for (lower upper size) in dimensions)
          (initial remainder i)
    (do
        (msg " " (+ lower (// remainder size) ) )
        (:= remainder (\ remainder size) ) ) ) )


(defun sim.real:imag:offset:value ( real-base imag-base offset )
    (if imag-base (then
        `(,(sim.memory:[] (+ real-base offset) )
          ,(sim.memory:[] (+ imag-base offset) ) ) )
    (else
        (sim.memory:[] (+ real-base offset) ) ) ) )



(def-sharp-sharp r
    `(sim.sharp-sharp-r ',(read) ',(read) ) )

(defun sim.sharp-sharp-r ( me-name register )
    (let ( (me (name:me me-name) ) )
        `(,([] (me:values      me) register)
          ,([] (me:value-names me) register) ) ) )
```

```
;***----------------------------------------------------------------********
;*** Copyright (C) 1988 John R. Ellis.  This file may be used for    ********
;*** educational and research purposes only, by the faculty, students, ********
;*** and staff of Yale University only.                              ********
;***----------------------------------------------------------------********
;==========================================================================
;
; TEST BED
;
; This module implements a test bed that allows the use of "trace
; snaphsots" to simulate the compiler interface without actually requiring
; the compiler to be present.
;
; To use it, first create a trace snapshot as described in
; GENERATE-CODE-HOOK.  Then, if the snapshot was for trace 3 of program
; PRIME, load the file PRIME-3.TRACE.
;
; (TLS NAME) [Test List Scheduler]
;     Invokes the test bed on the variable "name" that was loaded from the
;     file name.TRACE.
;
; The interface functions START-TRACE, PREDECESSORS, NAME:DATATYPE, and
; NAME:RANK are defined here that simulate the real ones.
;
; To get back to the "real" interface, call (TEST-BED.UNDO)
;
;==========================================================================

(declare (special
    *tb.symbol-table*        ;*** A symbol table (list) of tuples of the form:
                             ;***
                             ;***    (name datatype rank)

    *tb.index:datum*         ;*** A vector filled in by PREDECESSORS that
                             ;*** maps integer trace positions of operations
                             ;*** onto the client-supplied datum passed
                             ;*** into PREDECESSORS.

    *tb.index:result*        ;*** A vector mapping trace positions of
                             ;*** operations onto the original result
                             ;*** returned by PREDECESSORS (in the real
                             ;*** compiler).

    *tb.current-index*       ;*** The trace position of the next expected
                             ;*** operation expected by PREDECESSORS.  We
                             ;*** assume the client calls PREDECESSORS in
                             ;*** order.

    *tb.saved-names&defs*    ;*** List of pairs of function names and
                             ;*** definitions for the original functions
                             ;*** simulated by this interface.
    ) )

(if (! (boundp '*tb.saved-names&defs*) ) (then
    (:= *tb.saved-names&defs*
        (loop (for name in '(start-trace predecessors name:datatype name:rank))
        (save
            `(,name ,(fundef name) ) ) ) ) ) )


;***=======================================================================
;***
;*** (TLS NAME)
```

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

(defun tls
       ( (symbol-table live-before source-record-list live-after
          predecessors-results)
         &optional (print? t) )

        ;*** Remember the symbol table.
        ;***
    (:= *tb.symbol-table* symbol-table)

        ;*** Initialize the vectors mapping trace positions.
        ;***
    (:= *tb.index:datum*  (makevector (length source-record-list) ) )
    (:= *tb.index:result* (makevector (length source-record-list) ) )

    (vector:initialize *tb.index:result* predecessors-results)

        ;*** Initialze the starting index to 0, 1 if there is a DEF
        ;*** in the first position (the client doesn't call PREDECESSORS
        ;*** on DEF.
        ;***
    (:= *tb.current-index*
        (if (== 'def (oper:operator (car (car source-record-list) ) ) )
            1
            0) )

        ;*** Now invoke whatever we want to invoke.
        ;***
    (generate-code live-before source-record-list live-after)

    (if print? (then
        (sch.print-schedule) ) )
    () )


;***=======================================================================
;***
;*** (START-TRACE)
;***
;***=======================================================================

(defun start-trace ()
    () )


;***=======================================================================
;***
;*** (PREDECESSORS SOURCE-OPER TRACE-DIRECTION DATUM)
;***
;***=======================================================================

(defun predecessors ( source-oper trace-direction datum )
    (let*( (result
                (loop (for (index . reasons) in ([] *tb.index:result*
                                                    *tb.current-index*) )
                (save
                    `(,([] *tb.index:datum* index)
                      ,,reasons) ) ) ) )

        (:= ([] *tb.index:datum* *tb.current-index*) datum)
        (:= *tb.current-index* (+ &&& 1) )
```

```
        result) )

;***==========================================================================
;***
;*** (NAME:DATATYPE NAME)
;***
;***==========================================================================

(defun name:datatype ( name )
    (let ( ( (() datatype rank) (assoc name *tb.symbol-table*) ) )
        (assert datatype "NAME:DATATYPE: Name not found: " (h name) )
        datatype) )


(defun name:rank ( name )
    (let ( ( (() datatype rank) (assoc name *tb.symbol-table*) ) )
        (assert rank "NAME:DATATYPE: Name not found: " (h name) )
        rank) )


;***==========================================================================
;***
;*** (TEST-BED.UNDO)
;***
;***==========================================================================

(defun test-bed.undo ()
    (loop (for (name def) in *tb.saved-names&defs*) (do
        (:= (fundef name) def) ) )
    () )
```

```
;***------------------------------------------------------------------********
;*** Copyright (C) 1988 John R. Ellis.  This file may be used for     ********
;*** educational and research purposes only, by the faculty, students, ********
;*** and staff of Yale University only.                               ********
;***------------------------------------------------------------------********
;==========================================================================
;
; VALUE NODES
;
; A VN (Value Node) represents one value (and possibly an operation) in the
; expression DAG for a trace.
;
;======================;
(def-struct vn         ;
                       ;
    number             ; A unique number; guarranteed to produce source
                       ; ordering of nodes in the trace.
                       ;
    type               ; One of {DEF, DEF1, OPERATION, USE, USE1, COPY,
                       ; PSEUDO-OP}
                       ;
    constant?          ; True if a constant DEF.
                       ;
    datatype           ; Datatype of this value, one of {INTEGER, FLOAT}
                       ;
    datum              ; The opaque bookkeepr datum handed to us by
                       ; GENERATE-CODE.
                       ;
    name               ; The variable name written by this node;  for
                       ; constant DEFs, the value of the constant.
                       ;
    oper               ; Source NADDR operation this VN represents.
                       ;
    operand-vns        ; List of VNs that are read by this VN.
                       ;
    constraining-vns   ; List of VNs that must precede this VN, but which
                       ; are not values read by this node.
                       ;
    constraining-delays ; A list of numbers corresponding to
                       ; :CONTRAINT-VNS; each number specifies how many
                       ; cycles later this VN must be done after the
                       ; corresponding constraining VN.
                       ;
    off-live-vns       ; List of VNs "conditionally read" by this VN
                       ; on the off-trace edge (if it is a jump).
                       ;
    reading-vns        ; List of VNs that read the value of this VN.
                       ;
    constrained-vns    ; List of VNs that are constrained by this VN (via
                       ; :CONSTRAINING-VNS).
                       ;
    off-live-reading-vns; List of VNs (jumps) that "off-live read" this
                       ; VN.
                       ;
    depth              ; A lower bound on the earliest this VN could be
                       ; scheduled.
                       ;
    height             ; A lower bound on the "time" of this VN from the
                       ; exit of the DAG, analagous to the :DEPTH.
                       ;
    me                 ; The functional unit or constant generator where this
                       ; VN will be computed.
                       ;
```

```
    likely-mes         ; List of MES, the current best guess as to where
                       ; this VN will be located or computed.
                       ;
    bug-cycle          ; The cycle in which BUG guessed this VN could be
                       ; be scheduled.
                       ;
    cycle              ; In list scheduling, the current earliest cycle this
                       ; VN can be scheduled.
                       ;
    scheduled-cycle    ; In list scheduling, the cycle this VN was actually
                       ; scheduled.
                       ;
    register-bank-me   ; The register location of this VN and the time
    register           ; it is available.  For USEis, this is the desired
    register-cycle     ; register location (if given).
                       ;
    read-cycle         ; The last cycle in which the value of this VN
                       ; was read.  This tells us the cycle in which
                       ; the value is available on a register-bank or
                       ; constant- generator "port".
                       ;
    readers-left       ; The number of successors of this VN that read the
                       ; value (including off-live readers) that haven't
                       ; been scheduled yet.
                       ;
    predecessors-left  ; The number of predecessors of this VN (operands
                       ; and constraining VNs) that haven't been scheduled
                       ; yet.
                       ;
    locations          ; For USEs and DEFs only, a list of pairs (ME
                       ; REGISTER) specifying the locations of the USE/DEF.
    )                  ;
;======================;
;
;
; VNs form their own OBJECT-UNIVERSE (see OBJECT-UNIVERSE.LSP).
;
; (VN.INITIALIZE)
;     Initializes this module, forgetting about all previous VNs.
;
; (VN:CREATE VN)
;     Records VN in the VN object universe.   The idiom for making a
;     new VN is:
;
;         (VN:CREATE (VN:NEW ...) )
;
; (VN:OPERATOR VN)
; (VN:GROUP    VN)
;     Accessing functions for the NADDR operation of the VN.
;
; (VN:DELAY VN)
;     For operation VNs, the delay of the assigned ME (in :ME); 0 for
;     all other VN types.
;
; (VN:CONSTRAINED-VN:DELAY VN CONSTRAINED-VN)
;     VN is on of the :CONSTRAINING-VNS of CONSTRAINEDS-VN.  Returns the
;     constraining delay between the two (if there are several delays
;     between the two, the largest is returned).
;
; (VN:INSERT-VNS VN PREDECESSOR-VNS)
;     VN should have exactly one operand.  The list PREDECESSOR-VNS is
;     spliced between that operand and VN, so that each VN in
;     PREDECESSOR-VNS now reads the operand, and VN now reads all of the
```

```
;     predecessors.
;
; (VN:SPLICE-VN VN NEW-VN READING-VNS OFF-LIVE-READING-VNS)
;     READING-VNS and OFF-LIVE-READING-VNS should be subsets of the
;     corresponding readers of VN.  NEW-VN is spliced so that it is a
;     reader of VN, with READING-VNS now reading NEW-VN instead of VN.
;     All the VNs in OFF-LIVE-READING-VNS are made to off-live read NEW-VN
;     instead of VN.
;
; (VN:DELETE VN)
;     Deletes a VN from the DAG and VN universe.  A VN can be deleted
;     only if it has no readers and no off-live readers.
;
; (VN:PRINT VN)
;     Prints VN in a pretty debugging format.
;
; ##VN I
;     Interactive syntax for accessing VN with number I.
;
;==========================================================================

(visible-fields vn number type name oper)


(eval-when (compile load)
    (include list-scheduler:declarations) )


(def-object-universe *ls.vn-universe*
    (object-name             vn)
    (mapping-type            numbered-objects)
    (object-number-function
        (lambda ( object )
            (vn:number object) ) )
    (set-object-number-function
        (lambda ( object number )
            (:= (vn:number object) number) ) )
    (initial-size 200) )


(def-sharp-sharp vn
    `(vn-universe:number:vn ,(read) ) )


(defun vn.initialize ()
    (vn-universe:initialize)
    () )


(defun vn:create ( vn )
    (if (! vn) (then
        (:= vn (vn:new) ) )
    (else
        (assert (vn:is vn) ) ) )
    (vn-universe:add vn)
    vn)


(defun vn:operator ( vn )
    (assert (vn:is vn) )
    (oper:operator (vn:oper vn) ) )
```

```
(defun vn:group ( vn )
    (assert (vn:is vn) )
    (oper:group (vn:oper vn) ) )


(defun vn:delay ( vn )
    (assert (vn:is vn) )
    (caseq (vn:type vn)
        (operation
            (assert (vn:me vn) )
            (me:delay (vn:me vn) ) )
        (t
            0) ) )


(defun vn:constrained-vn:delay ( vn constrained-vn )
    (assert (vn:is vn) )
    (loop (for constraining-vn    in (vn:constraining-vns    constrained-vn) )
          (for constraining-delay in (vn:constraining-delays constrained-vn) )
          (when (== constraining-vn vn) )
    (maximize constraining-delay) ) )


(defun vn:insert-vns ( vn predecessor-vns )
    (assert (== 1 (length (vn:operand-vns vn) ) ) )
    (let ( (operand-vn (car (vn:operand-vns vn) ) ) )
        (:= (vn:reading-vns operand-vn) (top-level-removeq vn &&&) )

        (loop (for predecessor-vn in predecessor-vns) (do
            (push (vn:reading-vns operand-vn) predecessor-vn)
            (:= (vn:operand-vns predecessor-vn) `(,operand-vn) )
            (:= (vn:reading-vns predecessor-vn) `(,vn) ) ) )

        (:= (vn:operand-vns vn) predecessor-vns)
        () ) )


(defun vn:splice-vn ( vn new-vn reading-vns off-live-reading-vns )
    (assert (subset? reading-vns          (vn:reading-vns          vn) ) )
    (assert (subset? off-live-reading-vns (vn:off-live-reading-vns vn) ) )
    (assert (|| reading-vns off-live-reading-vns) )

    (:= (vn:reading-vns vn) (set-diffq &&& reading-vns) )
    (push (vn:reading-vns vn) new-vn)

    (:= (vn:operand-vns new-vn) `(,vn) )
    (:= (vn:reading-vns new-vn) reading-vns)

    (loop (for reading-vn in reading-vns) (do
        (:= (vn:operand-vns reading-vn)
            (top-level-substq new-vn vn &&&) ) ) )

    (:= (vn:off-live-reading-vns vn) (set-diffq &&& off-live-reading-vns) )

    (:= (vn:off-live-reading-vns new-vn) off-live-reading-vns)

    (loop (for off-live-reading-vn in off-live-reading-vns) (do
        (:= (vn:off-live-vns off-live-reading-vn)
            (top-level-substq new-vn vn &&&) ) ) )
    () )
```

```
(defun vn:delete ( vn )
    (assert (&& (! (vn:reading-vns          vn) )
                (! (vn:off-live-reading-vns vn) )
                (! (vn:off-live-vns         vn) ) ) )

        ;*** Remove references to this VN from any VNs constrained by this
        ;*** one;  ugh, messy.
        ;***
    (loop (for constrained-vn in (vn:constrained-vns vn) ) (do
        (loop (for constraining-vn in (vn:constraining-vns    constrained-vn))
              (for delay           in (vn:constraining-delays constrained-vn))
              (when (!== constraining-vn vn) )
              (initial new-constraining-vns    ()
                       new-constraining-delays () )
        (do
            (push new-constraining-vns    constraining-vn)
            (push new-constraining-delays delay) )
        (result
            (:= (vn:constraining-vns    constrained-vn)
                (dreverse new-constraining-vns) )
            (:= (vn:constraining-delays constrained-vn)
                (dreverse new-constraining-delays) ) ) ) ) )

        ;*** Now removes references to this VN from constraining VNs.
        ;***
    (loop (for constraining-vn in (vn:constraining-vns vn) ) (do
        (:= (vn:constrained-vns constraining-vn)
            (top-level-removeq vn &&&) ) ) )

        ;*** Remove all references to VN from its operands.
        ;***
    (loop (for operand-vn in (vn:operand-vns vn) ) (do
        (:= (vn:reading-vns operand-vn)
            (top-level-removeq vn &&&) ) ) )

    (vn-universe:delete vn)
    () )


(defun vn:print ( vn )
    (assert (vn:is vn) )

    (msg 0 (vn:number vn) ": " )
    (caseq (vn:type vn)
        (def       (msg "def "  (vn:name vn) " ") )
        (defi      (msg "defi " (vn:name vn) " ") )
        (use       (msg "use "  (vn:name vn) " ") )
        (usei      (msg "usei " (vn:name vn) " ") )
        (operation (msg (vn:oper vn)         " ") )
        (pseudo-op (msg (vn:oper vn)         " ") )
        (copy      (msg "copy " (vn:oper vn) " ") )
        (t
            (error (list vn "VN:PRINT: Unknown VN:TYPE.") ) ) )
    (if (vn:constant? vn) (then
        (msg "c ") ) )
    (if (vn:datatype vn) (then
        (msg (vn:datatype vn) " ") ) )
    (msg t)

    (if (vn:operand-vns vn) (then
        (msg "    operand-vns:      ")
        (loop (for operand-vn in (vn:operand-vns vn) ) (do
            (msg (vn:number operand-vn) " ") ) )
```

```
        (msg t) ) )

    (if (vn:constraining-vns vn) (then
        (msg "    constraining-vns: ")
        (loop (for constraining-vn    in (vn:constraining-vns    vn) )
              (for constraining-delay in (vn:constraining-delays vn) )
        (do
            (msg (vn:number constraining-vn) "@" constraining-delay " ") ) )
        (msg t) ) )

    (if (vn:off-live-vns vn) (then
        (msg "    off-live-vns:     ")
        (loop (for off-live-vn in (vn:off-live-vns vn) ) (do
            (msg (vn:number off-live-vn) " ") ) )
        (msg t) ) )

    (if (vn:constrained-vns vn) (then
        (msg "    constrained-vns:  ")
        (loop (for constrained-vn in (vn:constrained-vns vn) ) (do
            (msg (vn:number constrained-vn) " ") ) )
        (msg t) ) )

    (if (vn:depth vn) (then
        (msg "    depth:            " (vn:depth vn) t) ) )

    (if (vn:cycle vn) (then
        (msg "    cycle:            " (vn:cycle vn) t) ) )

    (if (vn:bug-cycle vn) (then
        (msg "    bug-cycle:        " (vn:bug-cycle vn) t) ) )

    (if (vn:scheduled-cycle vn) (then
        (msg "    scheduled-cycle:  " (vn:scheduled-cycle vn) t) ) )

    (if (vn:me vn) (then
        (msg "    me:               "
             (me:name (vn:me vn) ) t) ) )

    (if (vn:likely-mes vn) (then
        (msg "    likely-mes:       " (h (me-list:name-list (vn:likely-mes vn)))
             t) ) )

    (if (vn:locations vn) (then
        (msg "    locations:        "
             (h (loop (for (me register) in (vn:locations vn) ) (save
                    `(,(me:name me) ,register) ) ) )
             t) ) )

    (if (vn:register-bank-me vn) (then
        (msg "    register:         "
             (me:name (vn:register-bank-me vn) ) " " (vn:register vn)
             t) ) )
    () )
```

```
;***--------------------------------------------------------------------********
;*** Copyright (C) 1983 John R. Ellis.  This file may be used for       ********
;*** educational and research purposes only, by the faculty, students, ********
;*** and staff of Yale University only.                                 ********
;***--------------------------------------------------------------------********
;===============================================================================
;
; WRITE-VNS
;
; This module provides the interface functions for printing out VN DAGs.
; See the similar modules in DRAWING: for details about how to get
; DAGs printed out.
;
; (WRITE-VNS &OPTIONAL (FILENAME "IN.DAG) DAG-COMMANDS)
;     Writes a description of the current VN DAG to FILENAME.  DAG-COMMANDS
;     is an optional list of assignment statements that will be recorded
;     with the DAG and EVALed by the DAG-drawing software; these
;     assignments control the formatting of the DAG.  See the variable
;     *WV.DEFAULT-DAG-COMMANDS* below.
;
;===============================================================================

(eval-when (compile eval)
    (build '(drawing:dag-input-node) ) )

(defvar *wv.default-dag-commands*
    '( (:= *dag:box-height*                    46)
       (:= *dag:node-width*                   100)
       (:= *dag:line-width*                    25)
       (:= *dag:box-width*                     80)
       (:= *dag:minimum-level-separation*      30)
       (:= *dag:ideal-text-lines-per-node*      4)
       (:= *dag:fold-text-lines?*             () )
       (:= *dag:critical-threshold*             0)
       (:= *dag:remove-non-critical-nodes&edges?*
                                              () )
       (:= *dag:shade-critical-nodes?*        () )
       (:= *dag:text-x-margin*                  3)
       (:= *dag:make-slides*?                 () ) ) )


(defun write-vns ( &optional (filename "in.dag") dag-commands)
    (let ( (filename (file-list:filename
                         (file-list:merge filename '(() () () "DAG") ) ) ) )
        (iota ( (file filename '(out newversion) ) ) (without file
            (msg (h '(,,*wv.default-dag-commands* ,,dag-commands)
                     100000 100000) t)
            (msg (h (wv.dag:graph) 100000 100000) t)
            (filename file) ) ) ) )


(defun wv.dag:graph ()
    (let ( (*wv.all-nodes* () ) )
        (loop (for-each-vn vn)
              (when (wv.vn:visible? vn) )
        (do
            (push *wv.all-nodes*
                  (caseq (vn:type vn)
                      ( (def use)
                          (wv.def-use-vn:create-node   vn) )
                      ( (defi usei)
                          (wv.defi-usei-vn:create-node vn) )
                      (operation
```

```
                          (wv.operation-vn:create-node vn) )
                      (copy
                          (wv.copy-vn:create-node      vn) ) ) ) ) )
        (result
            (dreverse *wv.all-nodes*) ) ) ) )


(defun wv.def-use-vn:create-node ( vn )
    (dag:input-node:new
        name    (vn:number vn)
        label   '(,(string-msg (vn:number vn) " " (vn:name vn) )
                  ,(me-list:name-list (vn:likely-mes vn) ) )
        style   'shading2
        child-edge-styles
                (wv.vn:child-edge-styles vn) ) )


(defun wv.defi-usei-vn:create-node ( vn )
    (dag:input-node:new
        name    (vn:number vn)
        label   '(,(string-msg (vn:number vn) " " (vn:name vn) )
                  ,(wv.vn:location vn) )
        style   'shading2
        child-edge-styles
                (wv.vn:child-edge-styles vn) ) )


(defun wv.operation-vn:create-node ( vn )
    (dag:input-node:new
        name    (vn:number vn)
        label   '(,(string-msg (vn:number vn) " " (me:name (vn:me vn) ) )
                  ,(vn:oper vn)
                  ,(wv.vn:location vn) )
        child-edge-styles
                (wv.create-chain
                    (me:delay (vn:me vn) )
                    'node
                    '(,(me:name (vn:me vn) ) )
                    ()
                    'normal
                    (wv.vn:child-edge-styles vn) ) ) )


(defun wv.copy-vn:create-node ( vn )
    (dag:input-node:new
        name    (vn:number vn)
        label   '(,(string-msg (vn:number vn) " " (wv.vn:me-name vn) )
                  copy
                  ,(wv.vn:location vn) )
        style   'shaded
        child-edge-styles
                (wv.vn:child-edge-styles vn) ) )


(defun wv.vn:me-name ( vn )
    (?( (vn:register-bank-me vn)
        (me:name (vn:register-bank-me vn) ) )
      ( (vn:me vn)
        (me:name (vn:me vn) ) )
      (t
        "") ) )
```

```
(defun wv.vn:location ( vn )
    (?( (vn:register-bank-me vn)
        (string-msg (me:name (vn:register-bank-me vn) ) " "
                    (vn:register vn) ) )
      ( (vn:me vn)
        (me:name (vn:me vn) ) )
      (t
        "") ) )


(defun wv.vn:child-edge-styles ( vn )
    (nconc
        (loop (for reading-vn in (noduplesq (vn:reading-vns vn) ) )
              (when (wv.vn:visible? reading-vn) )
        (save
            (wv.vn:child-vn:style:create-edge vn reading-vn 'thick) ) )

        (loop (for constrained-vn in (noduplesq (vn:constrained-vns vn) ) )
              (when (&& (> (vn:constrained-vn:delay vn constrained-vn) 0)
                        (wv.vn:visible? constrained-vn) ) )
        (save
            (wv.vn:child-vn:style:create-edge vn constrained-vn 'shaded)))))


(defun wv.vn:child-vn:style:create-edge ( vn child-vn style )
    (let ( (next-cycle  (wv.vn:next-cycle vn) )
           (child-cycle (wv.vn:cycle child-vn) ) )

        (if (== next-cycle child-cycle) (then
            '(,(vn:number child-vn) ,style) )
        (else
            (let ( (closest-pred-vn (wv.vn:closest-pred-vn child-vn) ) )
                (if (!== vn closest-pred-vn) (then
                    '(,(vn:number child-vn) ,style) )
                (else
                    (car (wv.create-chain
                            (- child-cycle next-cycle)
                            'line
                            ()
                            style
                            style
                            '( (,(vn:number child-vn) ,style))))))))))


(defun wv.vn:closest-pred-vn ( vn )
    (let ( (closest-operand-vn
                (loop (for operand-vn in (vn:operand-vns vn) )
                      (maximize operand-vn (wv.vn:next-cycle operand-vn) ) ) )

           (closest-constraining-vn
                (loop (for constraining-vn    in (vn:constraining-vns    vn) )
                      (for constraining-delay in (vn:constraining-delays vn) )
                      (when (> constraining-delay 0) )
                (maximize constraining-vn
                          (wv.vn:next-cycle constraining-vn) ) ) ) )

        (if (&& closest-constraining-vn
                (> (wv.vn:next-cycle closest-constraining-vn)
                   (wv.vn:next-cycle closest-operand-vn) ) )
            closest-constraining-vn
            closest-operand-vn) ) )
```

```
(defun wv.create-chain
       ( length type label box-style edge-style child-edge-styles )
    (loop (initial prev-child-edge-styles child-edge-styles)
          (decr i from length to 1)
          (bind node (dag:input-node:new
                        name    (gensym)
                        type    type
                        label   label
                        style   box-style
                        child-edge-styles
                                prev-child-edge-styles) )
    (do
        (push *wv.all-nodes* node)
        (:= prev-child-edge-styles '( (,(dag:input-node:name node)
                                        ,edge-style) ) ) )
    (result prev-child-edge-styles) ) )


(defun wv.vn:next-cycle ( vn )
    (+ 1 (+ (wv.vn:cycle vn)
            (caseq (vn:type vn)
                 (operation (me:delay (vn:me vn) ) )
                 (t          0) ) ) ) )


(defun wv.vn:cycle ( vn )
    (caseq (vn:type vn)
        (def
            (if (== 'def1 (vn:type (car (vn:reading-vns vn) ) ) )
                -2
                -1) )
        (def1
            -1)
        ( (copy operation)
            (vn:cycle vn) )
        (use1
            (vn:cycle vn) )
        (use
            (if (== 'use1 (vn:type (car (vn:operand-vns vn) ) ) )
                (+ 1 (vn:cycle vn) )
                (vn:cycle vn) ) )
        (t
            (error (list vn "Case error.") ) ) ) )


(defun wv.vn:visible? ( vn )
    (caseq (vn:type vn)
        (pseudo-op
            () )
        (def
            (for-some (reading-vn in (vn:reading-vns vn) )
                (&& (!== 'use (vn:type reading-vn) )
                    (wv.vn:visible? reading-vn) ) ) )
        (def1
            (for-some (reading-vn in (vn:reading-vns vn) )
                (!== 'use1 (vn:type reading-vn) ) ) )
        ( (operation copy)
            t)
        ( (use1 use)
            (wv.vn:visible? (car (vn:operand-vns vn) ) ) )
        (t
            (error (list vn "Case error.") ) ) ) )
```