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

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

    ideal-code-generator:schedule
    ideal-code-generator:machine-description
    ) )

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

```
;===========================================================================
;
; This module provides a "description" of the ideal machine.  For now,
; that consists only of resource vectors.
;
;===========================================================================

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


(declare (special *operator:resource-group*) )
(:= *operator:resource-group* '(

    (goto        goto         )

    (assign      assign       )
    (vload       assign       )
    (vstore      assign       )

    (inot        logical      )
    (iand        logical      )
    (ior         logical      )

    (iadd        iadd         )
    (isub        iadd         )

    (imul        imul         )

    (idiv        idiv         )

    (fadd        fadd         )
    (fsub        fadd         )

    (fmul        fmul         )

    (fdiv        fdiv         )

    (ieq         comparison   )
    (feq         comparison   )
    (ine         comparison   )
    (fne         comparison   )
    (igt         comparison   )
    (fgt         comparison   )
    (ige         comparison   )
    (fge         comparison   )
    (ile         comparison   )
    (fle         comparison   )
    (ilt         comparison   )
    (flt         comparison   )
    (imin        comparison   )
    (fmin        comparison   )
    (imax        comparison   )
    (fmax        comparison   )
    (iabs        comparison   )
    (fabs        comparison   )

    (truego      cond-jump    )
    (falsego     cond-jump    )
    (if-ieq      cond-jump    )
    (if-feq      cond-jump    )
    (if-ine      cond-jump    )
    (if-fne      cond-jump    )
```

```
    (if-igt      cond-jump    )
    (if-fgt      cond-jump    )
    (if-ige      cond-jump    )
    (if-fge      cond-jump    )
    (if-ile      cond-jump    )
    (if-fle      cond-jump    )
    (if-ilt      cond-jump    )
    (if-flt      cond-jump    )
    ) )


(declare (special *resource-group:resource-vec*) )
(:= *resource-group:resource-vec* '(
    (goto          ( 0 ) )
    (assign        ( 0 ) )
    (logical       ( 0 ) )
    (iadd          ( 0 ) )
    (imul          ( 0 ) )
    (idiv          ( 0 ) )
    (fadd          ( 0 ) )
    (fmul          ( 0 ) )
    (fdiv          ( 0 ) )
    (comparison    ( 0 ) )
    (cond-jump     ( 0 ) )
    (miscellaneous ( 0 ) )
    ) )


;***========================================================================
;***
;*** (OPER:RESOURCE-VEC OPER)
;***
;*** Returns the resource vector of an operation.
;***
;***========================================================================

(defun oper:resource-vec ( oper )
    (let ( (resource-group (operator:resource-group (oper:operator oper) ) ) )
        (cadr (assoc resource-group *resource-group:resource-vec*) ) ) )


;***========================================================================
;***
;*** (OPERATOR:RESOURCE-GROUP OPERATOR)
;***
;*** Returns the resource group of an operator.
;***
;***========================================================================

(defun operator:resource-group ( operator )
    (let ( ( (() resource-group)
             (assoc operator *operator:resource-group*) ) )
        (|| resource-group
            'miscellaneous) ) )


;***========================================================================
;***
;*** (LIST-VECTOR-SUM V1 V2)
;***
;*** Sums up two lists as "vectors".  If one is longer than the other, it is
;*** padded with 0s.
;***
```

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

(defun list-vector-sum ( v1 v2 )
    (loop (initial rest-v1 v1
                   rest-v2 v2)
    (while (|| rest-v1 rest-v2) )
    (save
        (+ (if rest-v1 (car rest-v1) 0)
          (if rest-v2 (car rest-v2) 0) ) )
    (next rest-v1 (cdr rest-v1)
          rest-v2 (cdr rest-v2) ) ) )
```

```
;===============================================================================
;
; Ideal Code Generator Scheduler
;
; This module implements a code generator for the "ideal" machine --
; parallel NADDR (infinite registers, as many NADDR operations per cycle
; as specified in MACHINE-DESCRIPTION, usually infinite).
;
;===============================================================================

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

;***============================================================================
;***
;*** A TRACE-ELEMENT represents all the information about a single element
;*** of a trace.
;***
;***==========================
                          ;***
(def-struct trace-element ;***
    source                ;*** NADDR source instruction
                          ;***
    trace-direction       ;*** For conditional jumps only, the direction
                          ;*** that the trace takes (RIGHT or LEFT).
                          ;***
    bookkeeper-record     ;*** Bookkeeper token handed us (we don't look
                          ;*** at it)
                          ;***
    trace-position        ;*** Position on the original trace.
                          ;***
    (successors           ;*** The successors of this element on the data
        () suppress)      ;*** precedence DAG; a list of TRACE-ELEMENTs.
                          ;***
    reasons               ;*** List of the types of conflict between each
                          ;*** each successor and this element; one of
                          ;*** either OPERAND- or
                          ;*** POSSIBLE-OPERAND-CONFLICT, or
                          ;*** CONDITIONAL-CONFLICT.
                          ;***
    (predecessors         ;*** The predecessors of this element on the
                          ;*** data precedence DAG; a list of
        () suppress)      ;*** TRACE-ELEMENTs.
                          ;***
    (pred-distances       ;*** List corresponding to :PREDECESSORS, each
        () suppress)      ;*** element the "distance" of the corresponding
                          ;*** predecessor from this element.  A distance
                          ;*** of 5 means that predecessor must be
                          ;*** scheduled at least 5 cycles earlier than
                          ;*** this element.
                          ;***
    (num-preds-left 0)    ;*** Number of predecessors left unscheduled
                          ;*** (for consistency check only).
                          ;***
    (depth 0)             ;*** Depth of this element in the data precedence
                          ;*** DAG.
                          ;***
    (height 0)            ;*** Height of this element in the data
                          ;*** precedence DAG.
                          ;***
    priority              ;*** Scheduling priority of this element.
                          ;***
    release-time          ;*** Earliest cycle at which this could be
```

```
                          ;*** scheduled.
                          ;***
    cycle                 ;*** The cycle number that this element has
                          ;*** been scheduled in.
    )                     ;***
;***==========================
;***============================================================================


;***============================================================================
;***
;*** Miscellaneous global variables that should be declare somewhere.
;***
;***============================================================================

(declare (special
    *tr.space-mode*                 ;*** from the trace picker
    *tr.trace-picker*               ;*** from the trace picker
    *tr.window*                     ;*** from the trace picker

    *sch.critical-path-length*      ;*** critical path length of the current
                                    ;*** trace's DAG.
    *sch.cond-jump-count*           ;*** number of cond jumps in the current
                                    ;***

    *tr.dag-hook*                   ;***
    *tr.generate-code-hook*         ;***
    ) )

;***============================================================================
;***
;*** *SCH.TRACE-ELEMENTS*    is the list of all the trace element records.
;*** *SCH.MAX-SCHEDULE-SIZE* is the maximum size of the schedule.
;*** *SCH.SCHEDULE-SIZE*     is the size of the current schedule.
;*** *SCH.SCHEDULE*          is the array of trace elements in the schedule,
;***                         indexed by cycle.
;*** *SCH.RESOURCES*         is the array of resources used by the elements,
;***                         indexed by cycle.
;***                         in each cycle.
;*** (INITIALIZE-CODE-GENERATOR)
;*** (SCH.SCHED.INITIALIZE)  re-initializes the schedule.
;***
;***============================================================================

(declare (special
    *sch.trace-elements*
    *sch.schedule*
    *sch.resources*
    *sch.max-schedule-size*
    *sch.schedule-size*

    ) )


(defun initialize-code-generator ()
    (sch.sched-initialize) )

(defun sch.sched-initialize ()
    (:= *sch.trace-elements* () )
    (:= *sch.schedule-size*  0)
    (if (! (boundp '*sch.max-schedule-size*) ) (then
        (:= *sch.max-schedule-size* 200) ) )
    (if (|| (! (boundp '*sch.schedule*) )
            (!= *sch.max-schedule-size* (vectorlength *sch.schedule*) ) )
    (then
        (msg 0 "SCHEDULE: Re-intialize the schedule to a maximum size of "
              *sch.max-schedule-size* " elements." t)
        (:= *sch.schedule*  (makevector *sch.max-schedule-size*) )
        (:= *sch.resources* (makevector *sch.max-schedule-size*) ) )
    (else
        (loop (incr i from 0 to (+ -1 *sch.max-schedule-size*) ) (do
            (:= ([] *sch.schedule*  i) () )
            (:= ([] *sch.resources* i) () ) ) ) ) ) )


;***==========================================================================
;***
;*** (GENERATE-CODE BEFORE-LIVE SOURCE-RECORD-LIST AFTER-LIVE)
;***
;*** As documented in DOC:CODE-GEN-INTERFACE.DOC.
;***
;***==========================================================================

(defun generate-code ( before-live source-record-list after-live )
    (if *tr.generate-code-hook*
        (funcall *tr.generate-code-hook* source-record-list) )

    (sch.sched-initialize)

    (:= *sch.trace-elements*
        (sch.convert-to-trace-elements source-record-list) )

    (sch.build-the-dag                *sch.trace-elements*)
    (sch.set-heights-and-depths       *sch.trace-elements*)
    (sch.set-release-times            *sch.trace-elements*)
    (sch.assign-priorities            *sch.trace-elements*)
    (:= *sch.trace-elements*
        (sch.top-sort-by-priorities *sch.trace-elements*) )

    (if *tr.dag-hook*
        (funcall *tr.dag-hook*
                 (|| *sch.trace-elements* 'empty-trace) ) )

    (sch.schedule                     *sch.trace-elements*)
    'ideal-code-generator-schedule-dummy)


;***==========================================================================
;***
;*** (SCHEDULE:LENGTH SCHEDULE)
;*** (SCHEDULE:[]     SCHEDULE I)
;*** (SCHEDULE:JOIN   SCHEDULE I)
;*** (SCHEDULE:SPLIT  SCHEDULE I JUMP-NUMBER)
;***
;*** As documented in DOC:CODE-GEN-INTERFACE.DOC.
;***
;***==========================================================================

(defun schedule:length ( schedule )
    *sch.schedule-size*)

(defun schedule:[] ( schedule i )
    (for (elem in ([] *sch.schedule* (+ -1 i) ) ) (save
        `(,(trace-element:source             elem)
          ,(trace-element:bookkeeper-record elem) ) ) ) )
```

```
(defun schedule:join  ( schedule i )
    `( (use (,(gensym) ,i) )
       () ) )

(defun schedule:split ( schedule i jump-number )
    `( (def (,(gensym) ,i ,jump-number) )
       () ) )


;***==========================================================================
;***
;*** (SCHEDULE:PRINT SCHEDULE)
;***
;*** Prints out the schedule in a pretty way.
;***
;***==========================================================================

(defun schedule:print ( schedule )
    (msg 0 t)
    (loop (incr i from 0 to (+ -1 *sch.schedule-size*) ) (do
        (msg (j (+ i 1) 3) "] "
             (h (for (elem in ([] *sch.schedule* i) ) (save
                    (trace-element:source elem) ) ) )
             t) ) ) )


;***==========================================================================
;***
;*** (SCH.CONVERT-TO-TRACE-ELEMENTS SOURCE-RECORD-LIST)
;***
;*** Converts a list like that received by GENERATE-CODE, with the elements
;*** of the form:
;***
;***    (SOURCE TRACE-DIRECTION BOOKKEEPER-RECORD LIVE-OFF)
;***
;*** into a list of TRACE-ELEMENTS containing the appropriate information.
;***
;***==========================================================================

(defun sch.convert-to-trace-elements ( source-record-list )
    (loop (initial result ()
                   i      0)
    (for (source trace-direction bookkeeper-record live-off) in
         source-record-list)
    (when (|| (! (oper:property? source 'pseudo-op) )
              (memq (oper:operator source) '(def-block dcl esc assert) ) ) )
    (do
        (push result
              (trace-element:new source            source
                                 trace-direction   trace-direction
                                 bookkeeper-record bookkeeper-record
                                 trace-position    i) )
        (:= i (+ i 1) ) )
    (result (dreverse result) ) ) )

;***==========================================================================
;***
;*** (SCH.BUILD-THE-DAG TRACE-ELEMENTS)
;***
;*** Builds the DAG representing the data precedence graph of the trace
;*** in TRACE-ELEMENTS.  Calls the disambiguator interface (see
;*** DOC:DISAMB.DOC) to determine the data precedence relations.
;***
```

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

(defun sch.build-the-dag ( trace-elements )

            ;*** Tell the disambiguator that the compactor is about to
            ;*** start picking a new trace from NADDR program.  The
            ;*** individual operations of the trace are presented via
            ;*** the function PREDECESSORS.

    (start-trace)

            ;*** For each element of trace, hand it to PREDECESSORS and
            ;*** get back the lists of equal and strict predecessors.
            ;*** Add in corresponding edges between the trace elements.

    (loop (initial strict-pred&reason-list ()
                   equal-pred&reason-list  () )
    (for elem in trace-elements)
    (do

        (desetq (equal-pred&reason-list strict-pred&reason-list)
                (sch.get-predecessors elem) )

                ;*** If either we're saving space by generating no split
                ;*** copies, or if we're doing basic-block compaction
                ;*** only, we want to stop cond-jumps from going ahead
                ;*** of earlier trace elements by crocking up equal edges
                ;*** from the jumps to all previous elements in the trace.
                ;*** If we're just preserving source order of cond-jumps,
                ;*** we put equal edges between each cond-jump.
                ;
        (if (&& (oper:property? (trace-element:source elem) 'conditional-jump)
                (|| (== *tr.space-mode*   'nsc)
                    (== *tr.space-mode*   'cjo)
                    (== *tr.trace-picker* 'bb) ) )
        (then
            (loop (for prev-elem in trace-elements)
                  (while (!== prev-elem elem) )
            (do
                (if (&& (! (assoc prev-elem strict-pred&reason-list) )
                        (|| (!== *tr.space-mode* 'cjo)
                            (oper:property? (trace-element:source prev-elem)
                                            'conditional-jump) ) )
                (then
                    (push equal-pred&reason-list '(,prev-elem () ) ) ) )))))


                ;*** for each strict predecessor, make the predecessor point
                ;*** this element, this element point at the predecessor, and
                ;*** record the distance between the two as 1.
                ;
        (for ( (pred-elem reason) in strict-pred&reason-list) (do
            (push (trace-element:successors     pred-elem) elem)
            (push (trace-element:reasons        pred-elem) reason)
            (push (trace-element:predecessors   elem)      pred-elem)
            (push (trace-element:pred-distances elem)      1)
            (++ (trace-element:num-preds-left elem) ) ) )

                ;*** do the same thing for the equal predecessors, except
                ;*** that the distance is 0.
                ;
        (for ( (pred-elem reason) in equal-pred&reason-list) (do
            (push (trace-element:successors     pred-elem) elem)
```

```
            (push (trace-element:reasons        pred-elem) reason)
            (push (trace-element:predecessors   elem)      pred-elem)
            (push (trace-element:pred-distances elem)      0)
            (++ (trace-element:num-preds-left elem) ) ) )

        ) ) )


;***========================================================================
;***
;*** (SCH.GET-PREDECESSORS ELEM)
;***
;*** Gives the disambiguator the next ELEMent in the trace, and asks for
;*** its predecessors.  Returns a 2-element list:
;***
;***     (EQUAL-PRED&REASON-LIST STRICT-PRED&REASON-LIST)
;***
;*** Both sublists are lists of pairs of the form:
;***
;***     (PRED-ELEM REASON)
;***
;*** where PRED-ELEM is a predecessor and REASON is one of
;*** OPERAND-CONFLICT, POSSIBLE-OPERAND-CONFLICT, or CONDITIONAL-CONFLICT.
;***
;***========================================================================

(defun sch.get-predecessors ( elem )
    (let ( (equal-pred&reason-list  () )
           (strict-pred&reason-list () )
           ( predecessors-result
             (predecessors (trace-element:source          elem)
                           (trace-element:trace-direction elem)
                           elem) ) )

        (for ( (pred-elem reason elem-operand elem-type pred-operand pred-type)
               in predecessors-result)
        (do
            (if (sch.equal-predecessor? reason elem-type pred-type) (then
                (push equal-pred&reason-list '(,pred-elem ,reason) ) )
            (else
                (push strict-pred&reason-list '(,pred-elem ,reason) ) ) ) ) )

        '(,equal-pred&reason-list ,strict-pred&reason-list) ) )


;***========================================================================
;***
;*** (SCH.EQUAL-PREDECESSOR? REASON ELEM-TYPE PRED-TYPE)
;***
;*** Returns true if REASON, ELEM-TYPE, and PRED-TYPE describe a predecessor
;*** that is  an "equal" predecessor (can be done in the same cycle).
;***
;*** REASON is one of OPERAND-, CONDITIONAL-, or POSSIBLE-OPERAND-CONFLICT.
;*** ELEM-TYPE and PRED-TYPE are one of READ, WRITTEN, or CONDITIONAL-READ.
;***
;*** Does an awful lot more than it has to, for consistency checking.
;***
;***========================================================================

(defun sch.equal-predecessor? ( reason elem-type pred-type )
    (caseq reason
        ( (operand-conflict possible-operand-conflict)
            (? ( (&& (== 'written elem-type)
                     (== 'written pred-type) )
                 () )
               ( (&& (== 'written elem-type)
                     (== 'read    pred-type) )
                 t )
               ( (&& (== 'read    elem-type)
                     (== 'written pred-type) )
                 () )
               ( t
                 (error (list reason elem-type pred-type
                         "SCH.GET-PREDECESSORS: Invalid operand types.")))))

        ( conditional-conflict
            (? ( (&& (== 'written          elem-type)
                     (== 'conditional-read pred-type) )
                 () )
               ( t
                 (error (list reason elem-type pred-type
                         "SCH.GET-PREDECESSORS: Invalid operand types.")))))
        ( t
          (error (list reason "SCH.EQUAL-PREDECESSOR?: Invalid REASON.")))))


;***============================================================================
;***
;*** (SCH.SCHEDULE TRACE-ELEMENTS)
;***
;*** Makes a schedule from TRACE-ELEMENTS (sorted in priority-topological
;*** order).  The elements are placed in the array *SCH.SCHEDULE* and
;*** the resources used by the elements in a cycle in the array
;*** *SCH.RESOURCES*.  Scheduling is done by taking each element in turn
;*** and finding the earliest possible cycle in which it could be
;*** scheduled.  This is done by starting at the release time of the
;*** element and searching forward until a resource-compatible cycle is
;*** found.
;***
;***============================================================================

(defun sch.schedule ( trace-elements )

            ;*** for each trace element (in priority-sorted topological order)
            ;*** place it on the schedule at the earliest time allowed.
    (for (elem in trace-elements) (do
        (assert (= 0 (trace-element:num-preds-left elem) ) )

        (loop (step cycle from (trace-element:actual-release-time elem) ) (do
            (if (sch.resource-compatible elem cycle) (then
                (sch.place-on-schedule elem cycle)
                (return () ) ) ) ) ) ) )

            ;*** sort the elements in each cycle by trace order;  the n-way
            ;*** jumps must be sorted in source (trace) order.
    (loop (step cycle from 0 to (+ -1 *sch.schedule-size*) ) (do
        (:= ([] *sch.schedule* cycle)
            (sort ([] *sch.schedule* cycle)
                  #'(lambda (elem1 elem2)
                        (< (trace-element:trace-position elem1)
                           (trace-element:trace-position elem2) ) ) ) ) ) )
    )
```

```
;***============================================================================
;***
;*** (SCH.PLACE-ON-SCHEDULE ELEM CYCLE)
;***
;*** Place a trace element on the schedule at cycle CYCLE.
;***
;***============================================================================

(defun sch.place-on-schedule ( elem cycle )
    (:= *sch.schedule-size* (max *sch.schedule-size* (+ 1 cycle) ) )

    (:= (trace-element:cycle elem) cycle)
    (push ([] *sch.schedule* cycle) elem)

    (:= ([] *sch.resources* cycle)
        (list-vector-sum ([] *sch.resources* cycle)
                         (trace-element:resource-vec elem) ) )

    (for (succ-elem in (trace-element:successors elem) ) (do
        (-- (trace-element:num-preds-left succ-elem) ) ) )
    () )

;***============================================================================
;***
;*** (SCH.ASSIGN-PRIORITES TRACE-ELEMENTS)
;***
;*** Assigns priorities to each of the trace elements, guarranteeing that
;*** each element has priority strictly less than its predecessors.
;***
;***============================================================================

(defun sch.assign-priorities ( trace-elements )
    (for (elem in trace-elements) (do
        (:= (trace-element:priority elem)
            (trace-element:height   elem) ) ) ) )


;***============================================================================
;***
;*** (SCH.TOP-SORT-BY-PRIORITIES TRACE-ELEMENTS)
;***
;*** Destructively sorts TRACE-ELEMENTS by priority order (the priorities
;*** guarrantee a topological order).
;***
;***============================================================================

(defun sch.top-sort-by-priorities  ( trace-elements )
    (sort trace-elements
          #'(lambda ( elem1 elem2 )
                (> (trace-element:priority elem1)
                   (trace-element:priority elem2) ) ) ) )


;***============================================================================
;***
;*** (SCH.SET-HEIGHTS-AND-DEPTHS TRACE-ELEMENTS)
;***
;*** Calculates the height and depth of every element, and also
;*** *sch.max-height*, *sch.max-depth*, *sch.critical-path-length*, and
;*** *sch.cond-jump-count*.
;***
;***============================================================================
   fun sch.set-heights-and-depths ( trace-elements )
    (:= *sch.critical-path-length*  0)
    (:= *sch.cond-jump-count*       0)

            ;*** for each element (in forward topological order), calculate
            ;*** the depth of the element as 1 + the maximum depth of its
            ;*** predecesssors.  Also count the number of conditional jumps,
            ;*** record the critical path length.

    (for (elem in trace-elements) (do
        (if (! (trace-element:predecessors elem) ) (then
            (:= (trace-element:depth elem) 0) )
        (else
            (:= (trace-element:depth elem)
                (+ 1 (loop (initial max-pred-depth 0)
                    (for pred-elem in (trace-element:predecessors elem) )
                    (do
                        (:= max-pred-depth
                            (max max-pred-depth
                                 (trace-element:depth pred-elem) ) ) )
                    (result max-pred-depth) ) ) ) ) )

        (:= *sch.critical-path-length*
            (max *sch.critical-path-length* (trace-element:depth elem) ) )

        (if (memq (oper:group (trace-element:source elem) )
                  '(cond-jump if-then-else) )
            (++ *sch.cond-jump-count*) ) ) )

            ;*** for each element (in reverse topological order), calculate
            ;*** the height of the element as 1 + the maximum height of
            ;*** all its successors.

    (:= trace-elements (dreverse trace-elements) )
    (for (elem in trace-elements) (do
        (if (! (trace-element:successors elem) ) (then
            (:= (trace-element:height elem) 0) )
        (else
            (:= (trace-element:height elem)
                (+ 1 (loop (initial max-succ-height 0)
                    (for succ-elem in (trace-element:successors elem) )
                    (do
                        (:= max-succ-height
                            (max max-succ-height
                                 (trace-element:height succ-elem) ) ) )
                    (result max-succ-height) ) ) ) ) ) ) )
    (:= trace-elements (dreverse trace-elements) )
    )


;***===========================================================================
;***
;*** (SCH.SET-RELEASE-TIMES TRACE-ELEMENTS)
;***
;*** Sets the :RELEASE-TIME of each trace element to be 0, unless we are
;*** doing "minimum-release-time" space saving, in which case the release
;*** time of conditional jumps is calculated according to an obscure
;*** formula (see the description of 'MRT space-saving mode).  To prevent
;*** jumps from gathering at the end, since we don't have 2**n way jumps
;*** yet, we make sure that there is room for each jump at the end.
;***
```

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

(defun sch.set-release-times ( trace-elements )
    (let ( (max-depth  0)
          (jumps-left *sch.cond-jump-count*) )

        (for (elem in trace-elements) (do

            (if (!== *tr.space-mode* 'mrt) (then
                (:= (trace-element:release-time elem) 0) )
            (else
                (:= max-depth (max max-depth (trace-element:depth elem) ) )

                (if (memq (oper:group (trace-element:source elem) )
                          '(cond-jump if-then-else) )
                (then
                    (:= jumps-left (+ -1 jumps-left) )
                    (:= (trace-element:release-time elem)
                        (max 0 (- (min max-depth
                                       (- *sch.critical-path-length*
                                          jumps-left) )
                                  *tr.window*) ) ) )
                (else
                    (:= (trace-element:release-time elem) 0) ) ) ) ) ) ) ) )


;***===========================================================================
;***
;*** (SCH.RESOURCE-COMPATIBLE ELEM CYCLE)
;***
;*** Returns true if trace-element ELEM is resource compatible with the
;*** elements already scheduled in cycle CYCLE.
;***
;***===========================================================================

(defun sch.resource-compatible ( elem cycle )
    (for-every (elem-resource  in (trace-element:resource-vec elem) )
               (cycle-resource in ([] *sch.resources* cycle) )
        (<= (+ elem-resource cycle-resource) 1.0) ) )

;***===========================================================================
;***
;*** (TRACE-ELEMENT:ACTUAL-RELEASE-TIME ELEM)
;***
;*** Returns the actual release time of an element by taking the maximum
;*** of its :RELEASE-TIME and and the times specified by its
;*** :PRED-DISTANCES (the time of each predecessor plus the distance from
;*** that predecessor that this element should be scheduled).
;***
;***===========================================================================

(defun trace-element:actual-release-time ( elem )
    (let ( (release-time (trace-element:release-time elem) ) )

        (for (pred-elem in (trace-element:predecessors   elem) )
             (distance  in (trace-element:pred-distances elem) )
        (do
            (:= release-time
                (max release-time
                     (+ (trace-element:cycle pred-elem) distance) ) ) ) )

        release-time) )
```

```
;***============================================================================
;***
;*** (TRACE-ELEMENT:RESOURCE-VEC ELEM)
;***
;*** Returns the resource vector of a trace element.
;***
;***============================================================================

(defun trace-element:resource-vec ( elem )
    (oper:resource-vec (trace-element:source elem) ) )
```

```
;***============================================================================
;***
;*** Sample hook functions for accessing the trace hooks in the compactor.
;*** Each hook function is called with () before compaction starts to
;*** initialize it.  Then it is called with each trace.
;***
;*** *TR.GENERATE-CODE-HOOK* is called with the same values that
;***                         GENERATE-CODE is.
;***
;*** *TR.DAG-HOOK*           is called with the top-sorted list of
;***                         TRACE-ELEMENTS constructed by the ideal
;***                         code-generator's GENERATE-CODE.
;***
;***============================================================================


(declare (special
    *tr.generate-code-hook*
    *tr.dag-hook*
    ) )


(defvar *st.all-traces* ())
(defvar *st.all-dags*   ())


(defun st.generate-code-hook ( trace )
    (if (! trace)
        (:= *st.all-traces* () )
        (:= *st.all-traces*
            (append1 *st.all-traces* trace) ) ) )

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


(defun st.dag-hook ( trace )
    (if (! trace)
        (:= *st.all-dags* () )
        (:= *st.all-dags*
            (append1 *st.all-dags* trace) ) ) )

(:= *tr.dag-hook* 'st.dag-hook)
```