(:: (build *lcg.build-module-list*)
(*** (build.compile *lcg.build-module-list*)
(= *lcg.build-module-list* '( (ideal-code-generator:schedule
        (ideal-code-generator:machine-description
))))
 (:: (build-module-list* (append *build-module-list* *lcg.build-module-list*)) )

PS:<C.S.BULLDOG.IDEAL-CODE-GENERATOR>BUILD.LSP.1
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 vload )
  (vstore vstore )
  (inot logical )
  (land logical )
  (lor logical )
  (iadd iadd )
  (isub isub )
  (iul iul )
  (idiv idiv )
  (fadd fadd )
  (fsub fsub )
  (faul faul )
  (fdlv fdlv )
  (leq comparison )
  (leq eq comparison )
  (lt eq comparison )
  (lte eq comparison )
  (ltf eq comparison )
  (lti eq comparison )
  (fle comparison )
  (lt comparison )
  (lim comparison )
  (fmin comparison )
  (fmax comparison )
  (flb comparison )
  (fabs comparison )
  (true go cond-jump )
  (false go cond-jump )
  (if-leq cond-jump )
  (if-ge cond-jump )
  (if-fge cond-jump )
  (if-le cond-jump )
  (if-ge cond-jump )
  (if-fge cond-jump )
  (if-le cond-jump )
  (if-ge cond-jump )
  (if-fge cond-jump )
)

(declare (special *resource-group:resource-vec*) )
(;; »resource-group:resource-vec* '(
  (goto goto 0 )
  (assign assign 0 )
  (logical logical 0 )
  (iadd iadd 0 )
  (isub isub 0 )
  (iul iul 0 )
  (idiv idiv 0 )
  (fadd fadd 0 )
  (fsub fsub 0 )
  (faul faul 0 )
  (fdlv fdlv 0 )
  (leq comparison 0 )
  (lt eq comparison 0 )
  (ltf eq comparison 0 )
  (lti eq comparison 0 )
  (fle comparison 0 )
  (lt comparison 0 )
  (lim comparison 0 )
  (fmin comparison 0 )
  (fmax comparison 0 )
  (flb comparison 0 )
  (fabs comparison 0 )
  (true go cond-jump)
  (false go cond-jump)
  (if-leq cond-jump)
  (if-ge cond-jump)
  (if-fge cond-jump)
  (if-le cond-jump)
)

== (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») )
    (assoc resource-group »resource-group:resource-group:resource-vec*») ) )

== (LIST-VECTOR-SUM V1 V2) ==

*** Sums up two lists as "vectors". If one is longer than the other, it is padded with 0s.
***

PS: <C.S.BULLDOG.IDEAL-CODE-GENERATOR>MACHINE-DESCRIPTION.LSP.7
(defun list-vector-sum (vl v2)
  (loop (initial rest-vl vl
           rest-v2 v2)
    (while (|| rest-vl rest-v2) )
    (save
     (+ (if rest-vl (car rest-vl) 0)
        (if rest-v2 (car rest-v2) 0))
     (next rest-vl (cdr rest-vl)
               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
  trace-direction
  bookkeeper-record
  trace-position
  (successors () suppress)
  reasons
  (predecessors () suppress)
  (pred-distances () suppress)
  (num-preds-left 0)
  (depth 0)
  (height 0)
  priority
  release-time

  *** WADDR source instruction
  *** For conditional jumps only, the direction that the trace takes (RIGHT or LEFT).
  *** Bookkeeper token handed us (we don't look at it)
  *** Position on the original trace.
  *** The successors of this element on the data precedence DAG; a list of TRACE-ELEMENTS.
  *** List of the types of conflict between each successor and this element; one of either OPERAND- or POSSIBLE-OPERAND-CONFLICT, or CONDITIONAL-CONFLICT.
  *** The predecessors of this element on the data precedence DAG; a list of TRACE-ELEMENTS.
  *** List corresponding to :PREDECESSORS, each 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.
  *** Number of predecessors left unscheduled (for consistency check only).
  *** Depth of this element in the data precedence DAG.
  *** Height of this element in the data precedence DAG.
  *** Scheduling priority of this element.
  *** Earliest cycle at which this could be scheduled.

  cycle
  *** The cycle number that this element has been scheduled in.
r

(!= *sch.max-schedule-size* (vectorlength »sch.schedule»))

(then
  (msg 0 "SCHEDULE: Re-initialize the schedule to a maximum size of "
  (+ *sch.schedule-size* " elements." t)
  (= »sch.schedule* (makevector *sch.max-schedule-size*) )
  (= »sch.resources* (makevector *sch.max-schedule-size*) ) )

(else
  (loop (incr 1 from 0 to (+ -1 *sch.max-schedule-size*) ) (do
    (= (j (+ 1 1) 3) "] " ) )
  ) )

odef (sch.trace-elements)

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

(defun schedule: [] ( achedule 1 )
  (for (elea in ( ] »sch.achedule« (+ -1 i) ) ) (aave
    (trace-element:source elea) )
  )

(defun schedule:split ( schedule 1 jump-number )
  (def (.[genys] ,1 ,jump-number)
    (do
      (msg ( ] (+ 1 1) 3) "] "
        (for (elem in ( ] »sch.achedule« 1 ) ) (save
          (trace-element:source elem) )
      )
    )
  )

(defun schedule:print ( schedule )
  (nsg 0 t)
  (loop (incr 1 from 0 to (+ -1 »sch.schedule-size») ) (do
    (asg (j (+ 1 1) 3) "] " )
  )
)

(defun sch.convert-to-trace-elements ( source-record-list )
  (loop (Initial result ()
    (for (source trace-direction bookkeeper-record live-off) in
      aource-record-liat)
    (when (|| (! (oper:property? source 'pseudo-op) )
      (neaq (oper:operator source) '(def-block del esc assert) ) )
    )
    (do
      (push result
        (trace-element:new source source
          *trace-direction* trace-direction
          trace-position 1) )
    )
  )
)

(defun schedule:join ( schedule 1 )
  '( (use (.[genys] ,1) )
    (] ) )

(defun schedule:print ( schedule )
  (nsg 0 t)
  (loop (incr 1 from 0 to (+ -1 »sch.schedule-size») ) (do
    (asg (j (+ 1 1) 3) "] " )
  )
)

(defun schedule:print ( schedule )
  (nsg 0 t)
  (loop (incr 1 from 0 to (+ -1 »sch.schedule-size») ) (do
    (asg (j (+ 1 1) 3) "] " )
  )
)

(defun sch.build-the-dag (trace-elements)
  (sch.top-sort-by-priorities *sch.trace-elements* )

(defun sch.schedule ( trace-elements )
  (ideal-code-generator-schedule-dUBBy)

;***==============================33==================3==3=====3==3=====:
;*** (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-inltialize)
  (= »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-hooks* (funcall *tr.dag-hook* "empty-trace) )
  (sch.schedule *sch.trace-elements*)

;**** (SCHEDULE:LENGTH SCHEDULE)
;****
;**** Prints out the schedule in a pretty way.
;****
;****
;
(defun schedule:length ( schedule )
  *sch.schedule-size*)

;**** (SCHEDULE:[] SCHEDULE I)
;****
;****
;
(defun schedule: [] ( achedule 1 )
  (for (elea in ( ] »sch.achedule« (+ -1 i) ) ) (aave
    (trace-element:source elea) )
  )

;**** (SCHEDULE:JOIN SCHEDULE I)
;****
;****
;
(defun schedule:split ( schedule 1 jump-number )
  (def (.[genys] ,1 ,jump-number)
    (do
      (msg ( ] (+ 1 1) 3) "] "
        (for (elem in ( ] »sch.achedule« 1 ) ) (save
          (trace-element:source elem) )
      )
    )
  )

;**** (SCHEDULE:SPLIT SCHEDULE I JUMP-NUMBER)
;****
;****
;
(defun schedule:print ( schedule )
  (nsg 0 t)
  (loop (incr 1 from 0 to (+ -1 »sch.schedule-size») ) (do
    (asg (j (+ 1 1) 3) "] " )
  )
)

;*** (SCHEDULE:PRINT SCHEDULE)
;***
;*** Prints out the schedule in a pretty way.
;***
;***
;
(defun schedule:print ( schedule )
  (nsg 0 t)
  (loop (incr 1 from 0 to (+ -1 »sch.schedule-size») ) (do
    (asg (j (+ 1 1) 3) "] " )
  )
)

;*** (SCHEDULE: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)
  (sch.top-sort-by-priorities *sch.trace-elements* )

;*** (SCHEDULE:LENGTH SCHEDULE)
;***
;***
;
(defun schedule:length ( schedule )
  *sch.schedule-size*)

;*** (SCHEDULE:[] SCHEDULE I)
;***
;***
;
(defun schedule: [] ( achedule 1 )
  (for (elea in ( ] »sch.achedule« (+ -1 i) ) ) (aave
    (trace-element:source elea) )
  )

;*** (SCHEDULE:JOIN SCHEDULE I)
;***
;***
;
(defun schedule:split ( schedule 1 jump-number )
  (def (.[genys] ,1 ,jump-number)
    (do
      (msg ( ] (+ 1 1) 3) "] "
        (for (elem in ( ] »sch.achedule« 1 ) ) (save
          (trace-element:source elem) )
      )
    )
  )
(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.

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

  (start-trace)

  (loop (initial strict-predicate-list ()
         equal-predicate-list ()
         (for elem in trace-elements)
           (do
             (desetq (equal-predicate-list strict-predicate-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 creating 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-node* 'noc)
                     (== *tr.space-node* 'cjo)
                     (== *tr.trace-picker* 'bb))
               (then
                 (loop (for prev-elem in trace-elements)
                       (while (&&! prev-elem elem))
                         (do
                           (if (&& (! (assoc prev-elem strict-predicate-list) )
                                 (condition:property? (trace-element:source prev-elem)
                                                       'conditional-jump)
                                 (then
                                   (push equal-predicate-list '(.prev-elem .reason) ) ) )
                           (else
                             (push strict-predicate-list '(.prev-elem .reason) ) ) )
                 '(.equal-predicate-list .strict-predicate-list) )
             )))

             ;; 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 elem-type pred-type) in strict-predicate-list)
               (do
                 (push (trace-element:successors pred-elem elem)
                       (trace-element:reasons pred-elem)
                       (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 elem-type pred-type) in equal-predicate-list)
               (do
                 (push (trace-element:successors pred-elem elem)
                       (trace-element:reasons pred-elem)
                       (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-PREDICATION-NLIST STRICT-PREDICATION-NLIST)
  ;; 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-predicate-list ()
            strict-predicate-list 0
            predecessors-result
            predecessors (predecessors (trace-element:source elem)
                                       (trace-element:trace-direction elem)
                                       elem) )
          (for ( (pred-elem reason elem-type pred-type) in predecessors-result)
            (do
              (if (sch.equal-predecessor? reason elem-type pred-type)
                (then
                  (push equal-predicate-list '(.pred-elem .reason) )
                  (else
                    (push strict-predicate-list '(.pred-elem .reason) )
                  ) )
              ))
          (,equal-predicate-list .strict-predicate-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)
      (caseq elem-type pred-type
        (read written))
      )))
(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:nua-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.place-on-schedule elem cycle)
            (return ()) ))
    )))

(SCH.ASSIGN-PRIORITIES TRACE-ELEMENTS)
*** Assigns priorities to each of the trace elements, guaranteeing 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 guarantee a topological order).

(defun sch.top-sort-by-priorities (trace-elements)
    (sort trace-elements "(lambda (elem 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.

(defun sch.set-heights-and-depths (trace-elements)
    ;* \(\text{lambdas (elem elem2)}
    ;** (trace-element:priority elem1) (trace-element:priority elem2))
    ;**
    ;** (sch.set-heights-and-depths)
fun sch.set-heights-and-depths ( trace-elements )
(type := *sch.critical-path-length* 0)
(type := *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
;*** predecessors. Also count the number of conditional jumps.
;*** Record the critical path length.
(for (elen in trace-elements) (do
  (if (! (trace-element:precedors elen) ) (then
    (type := (trace-element:depth elen) 0)
  )
  (else
    (type := (trace-element:depth elen)
      (+ 1 (loop (initial max-pred-depth 0)
        (for pred-elen in (trace-element:precedors elen) )
        (do
          (type := max-pred-depth
            (max max-pred-depth
              (trace-element:depth pred-elen) )
          )
        )
      )
    )
  )
  )
(type := *sch.critical-path-length* (max max *sch.critical-path-length* (trace-element:depth elen) )
)
(if (memq (oper:group (trace-element:source elen) )
  '(*cond-jump if-then-else* )
  then
    (type := Junps-left (+ -1 Junps-left) )
    (type := (trace-element:release-time elen)
      (+ 0 (- (aln max-depth
        (- *sch.critical-path-length* jumps-left) )
      )
    )
  )
  )
(if (memq (oper:group (trace-element:source elen) )
  '(*cond-jump if-then-else* )
  then
    (type := Junps-left (+ -1 Junps-left) )
    (type := (trace-element:release-time elen)
      (+ 0 (- (aln max-depth
        (- *sch.critical-path-length* jumps-left) )
      )
    )
  )
  )
)

defun sch.set-release-times ( trace-elements )
(let ( (max-depth 0)
  (Junps-left *sch.cond-jump-count* )
)
(for (elem in trace-elements) (do
  (if (! tr.space-node 'mrt) (then
  (type := (trace-element:release-time elem) 0)
  )
  (else
    (type := max-depth (max max-depth (trace-element:depth elem) )
  )
  )
  (if (memq (oper:group (trace-element:source elem) )
    '(*cond-jump if-then-else* )
    then
      (type := Junps-left (+ -1 Junps-left) )
      (type := (trace-element:release-time elem)
        (+ 0 (- (aln max-depth
          (- *sch.critical-path-length* jumps-left) )
        )
      )
    )
  )
  )
(type := (trace-element:release-time elem) 0) ) ) )

PS:<C.S.BULLDOG.IDEAL-CODE-GENERATOR>SCHEDULE.LSP.17
(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* (appendl *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* (appendl *st.all-dags* trace) ) ) )
  (:= *tr.dag-hook* 'st.dag-hook)