BOTTOM-UP GREEDY
(BUG.ASSIGN-MES)

This module assigns MESs to each VN, using an algorithm inspired by the top-down greedy register allocation. This algorithm is “bottom-up” because we draw our dag 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)
  (resources.initialize)
  (bug.assign-likely-mes)
  (loop (for vn in (bug.depth-sort-vns @ls.exit-vns)) (do (bug.vn:assign vn ()) ) )
  (bug.make-destinations) )
() )

---

*** (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 MESs 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 fn 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)) ) )
      ) )
    (error (list vn "BUG.VN:ASSIGN: Case error.")) ) ) ) )
() )
(defun bug.use-vn:assign (vn destinations)
  (bug.vn:assign (car (vn:operand-vns vn))
    (vn:likely-nes vn))
  (= (vn:bug-cycle vn) (vn:bug-cycle (car (vn:operand-vns vn)))))

(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
        operand-destinations) )
    (let ( (cycleftne-llst (bug.vn:destinations:cycleftne-llst vn destinations))
        (best-cycle ()
        (best-ne ()
        (desetq ( (best-cycle best-ne) . () ) cycleftne-llst))
      (else (desetq (best-cycle best-ne)
                      (loop (for cycle «*«
                              in cycleftne)
                            (bind (cycle ne) cycleftne)
                            (loop (for operand-vn
                                   in (vn:operand-vns vn)
                                   (reduce nax 0
                                (ne:ne:delay (vn:ne operand-vn) ne))))))))
  (bug.vn:cycle:ne:schedule vn best-cycle best-ne))

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

(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:numel-list
      (bug.vn:destinations:cycle:numel-list vn destinations)
      operand-destinations
      (if ((mesq operand-vn (vn:operand-vns vn))
        (then))
          (do (bug.vn:assign operand-vn operand-destinations) ))
    (let ( (cycle:numel-list (bug.vn:destinations:cycle:numel-list vn destinations))
      (best-cycle ()
      (best-me ())
      (if (! (cdr cycle:numel-list))
          (do (desetq (best-cycle best-me)
                      (loop (for cycle numel in cycle:numel-list)
                        (bind (cycle me) cycle#me)
                      (minimize cycle numel
                        (loop (for operand-vn in (vn:operand-vns vn)
                                (reduce nax 0
                              (me:me:delay (vn:me operand-vn) me)))))))))
      (else (desetq (best-cycle best-me)
                      (loop (for cycle numel in cycle numel-list)
                        (bind (cycle me) cycle#me)
                      (minimize cycle numel
                        (loop (for operand-vn in (vn:operand-vns vn)
                                (reduce nax 0
                              (me:me:delay (vn:me operand-vn) me)))))))))
  (bug.vn:cycle:me:schedule vn best-cycle best-me))

;; we make a "best guess" about the MEs likely to be used for VN; that
;; best guess is then based on DESTINATIONS for the recursive assign
;; 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.
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.)

(\(\text{defun bug.vn:destinations:cycleftne-llst (vn destinations)}\)
  (\(\text{loop (for \(\text{ne} \in (\text{vn:likely-nes vn}) \))}\)
    (\(\text{when (ne:datun:ok? ne vn) \)}\)
      (\(\text{bind cycle (bug.vn:ne:cycle vn me)}\)
        \(\text{cost (+ cycle \)}\)
        \(\text{(+ (ne:delay ne) \)}\)
        \(\text{(bug.ne:ne-Hst:Bin-delay ne destinations) \))\)
      \(\text{(save-minimum \('.\text{cycle ,ae) cost}') \))\)
    \)
  \)
)

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.)

(\(\text{defun bug.vn:destinations:cycle-me-list (vn destinations)}\)
  \(\text{(loop (for \(\text{me} \in (\text{vn:likely-me-es vn}) \))}\)
    \(\text{(bind cycle (bug.vn:me:cycle vn me)}\)
      \(\text{cost (+ cycle \)}\)
      \(\text{(+ (me:delay me) \)}\)
      \(\text{(bug.me:me-list:min-delay me destinations) \))\)
    \)
  \)
)

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.)

(\(\text{defun bug.vn:destinations:cycle-me-list (vn destinations)}\)
  \(\text{(loop (for \(\text{me} \in (\text{vn:likely-me-es vn}) \))}\)
    \(\text{(bind cycle (bug.vn:me:cycle vn me)}\)
      \(\text{cost (+ cycle \)}\)
      \(\text{(+ (me:delay me) \)}\)
      \(\text{(bug.me:me-list:min-delay me destinations) \))\)
    \)
  \)
)

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.)

(\(\text{defun bug.vn:destinations:cycle-me-list (vn destinations)}\)
  \(\text{(loop (for \(\text{me} \in (\text{vn:likely-me-es vn}) \))}\)
    \(\text{(bind cycle (bug.vn:me:cycle vn me)}\)
      \(\text{cost (+ cycle \)}\)
      \(\text{(+ (me:delay me) \)}\)
      \(\text{(bug.me:me-list:min-delay me destinations) \))\)
    \)
  \)
)

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.)

(\(\text{defun bug.vn:destinations:cycle-me-list (vn destinations)}\)
  \(\text{(loop (for \(\text{me} \in (\text{vn:likely-me-es vn}) \))}\)
    \(\text{(bind cycle (bug.vn:me:cycle vn me)}\)
      \(\text{cost (+ cycle \)}\)
      \(\text{(+ (me:delay me) \)}\)
      \(\text{(bug.me:me-list:min-delay me destinations) \))\)
    \)
  \)
)

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.)

(\(\text{defun bug.vn:destinations:cycle-me-list (vn destinations)}\)
  \(\text{(loop (for \(\text{me} \in (\text{vn:likely-me-es vn}) \))}\)
    \(\text{(bind cycle (bug.vn:me:cycle vn me)}\)
      \(\text{cost (+ cycle \)}\)
      \(\text{(+ (me:delay me) \)}\)
      \(\text{(bug.me:me-list:min-delay me destinations) \))\)
    \)
  \)
)

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.)

(\(\text{defun bug.vn:destinations:cycle-me-list (vn destinations)}\)
  \(\text{(loop (for \(\text{me} \in (\text{vn:likely-me-es vn}) \))}\)
    \(\text{(bind cycle (bug.vn:me:cycle vn me)}\)
      \(\text{cost (+ cycle \)}\)
      \(\text{(+ (me:delay me) \)}\)
      \(\text{(bug.me:me-list:min-delay me destinations) \))\)
    \)
  \)
)

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.)

(\(\text{defun bug.vn:destinations:cycle-me-list (vn destinations)}\)
  \(\text{(loop (for \(\text{me} \in (\text{vn:likely-me-es vn}) \))}\)
    \(\text{(bind cycle (bug.vn:me:cycle vn me)}\)
      \(\text{cost (+ cycle \)}\)
      \(\text{(+ (me:delay me) \)}\)
      \(\text{(bug.me:me-list:min-delay me destinations) \))\)
    \)
  \)
)

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.)

(\(\text{defun bug.vn:destinations:cycle-me-list (vn destinations)}\)
  \(\text{(loop (for \(\text{me} \in (\text{vn:likely-me-es vn}) \))}\)
    \(\text{(bind cycle (bug.vn:me:cycle vn me)}\)
      \(\text{cost (+ cycle \)}\)
      \(\text{(+ (me:delay me) \)}\)
      \(\text{(bug.me:me-list:min-delay me destinations) \))\)
    \)
  \)
)
(defun bug.depth-sort-vna ( vn-list )
  (sort vn-list
   (if (= (vn:depth vni) (vn:depth vn2))
    (lexorder (vn:name vni) (vn:name vn2))
    (t)))
)

(dfun bug.height-sort-vna ( vn-list )
  (sort vn-list
   (if (= (vn:height vni) (vn:height vn2))
    (lexorder (vn:name vni) (vn:name vn2))
    (t)))
)

;***========S======================3=33=3=====3====S3=3=====:
;***
;*** (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-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:me-list:min-delay ( me me-list )
  (if (null me-list) (then 0)
    (loop (for next-me in me-list)
      (reduce min 100000 (me:delay next-me me) ) )
  )
)

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

;else

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

(defun bug.me:me-list:min-delay ( me me-list )
  (if (null me-list) (then 0)
    (loop (for next-me in me-list)
      (reduce min 100000 (me:delay me next-me) ) )
  )
)
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 (and (vn:constant? vn)
           (vn:likely-use vn)
           (> (dag.name:trace-write-count (vn:name vn)) 0))
      (then

        (assert els.new-defs-allowed? "BUG: Huh?"
          (best-cycle 100000)
          (best-se))

        (loop (for me in (se:input destination))
          (when (and (= 'register-bank (me:type me))
                     (se: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-se me))))))))

* (BUG.MAKE-DEF1S&USES1S)

* This procedure is called after all main recursive assignment algorithms have been run. It inserts DEFs between DEFs and their readers, and USEs between USEs and their operands.

** First, USEs are inserted between USEs that have no given locations and their operands. Then DEFs are inserted between all DEFs and their readers. Finally, USEs are inserted between USEs that do have given locations and their operands.

(defun bug.make-def1s&uses1s ()
  ;*** Make USEs for USEs that have locations.
  (loop (for vn in *ls.exit-vns*)
    (when (and (= 'use (vn:type vn))
             (vn:locations vn))
      (do
        (bug.closed-use-vn:make-uses vn)
        (bug.def-vn:make-uses vn))))

* ;*** Make DEFs for DEFs.

* ;*** Make USEs for USEs that don't have locations.

* ;*** Make USEs for USEs that have locations.

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

1

2

PS: <C.S.BULLDOG.LIST-SCHEDULER.TEST> BUG-DEFS-USES.LSP .57
(defun bug.open-use-vn:make-use (vn)
  (let*( (operand-vn (car (vn:operand-vna vn)))
         (pred-vna
          (if (-- 'defl (vn:type operand-vn))
              (vn:reading-vna (car (vn:operand-vna operand-vn)))
              (vn:operand-vna vn))
          )
    (loop (for pred-vn in pred-vns)
      (bind usei-vn (vn:create (vn:new
                                type 'usei
                                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-nes (',me
                                             register-bank me
                                             register register) )))))))

(defun bug.open-use-vn:make-useis (vn)
  (list (vn:use-vns vn)
        (all-reading-vns
         (append (bug.height-sort-vns (vn:reading-vna vn))
                 (vn:off-live-reading-vns vn))
         (number-reading-vns)
         (length (vn:reading-vna 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)
  (bind reading-vn? (<= 1 number-reading-vns)
                   (bug.def-vn:reading-vna:choose-me
                    vn
                    (if reading-vn? reading-vn ())
                    )
     (do
      (if (! (nenq reading-vn? (vn:likely-nes vn))
        (push reading-vn? reading-vn)
      )
      (do
        (if (== reading-vn? (vn:likely-nes vn))
          (push (vn:likely-nes vn)
                (cq (me:type me)
                     (register-bank
                      (:= (me:registers-left me) (- &a 1)))
                     (constant-generator
                      (cycle:resource-request:schedule
                       (cycle:resource-request:first-available-cycle
                        0 me
                        (me:resources me)))
                      (me:resources me)))
          (t
           (error (list me "Case error.")
                   )))
        (let ( (ne-entry (assoc reading-vn? ne-table) )
               )
          (if (! ne-entry)
            (:= ne-entry '(.ne ()()))
            (push ne-table ne-entry)
          )
          (if reading-vn?
            (push (cadr ne-entry) reading-vn)
            (elae
             (push (caddr ne-entry) reading-vn)
            )
          )
        )
      )
    )
  )))

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

(loop (for (ne reading-vns off-live-reading-vns) in ne-table)
  (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-nes '(,ne)
    ne ne) )
  (do
    (if register (then
      (:= (vn:register-bank-ne defi-vn) ae)
      (: = (vn:register defi-vn) register) )
    ) )
  (vn:splice-vn vn defi-vn reading-vns off-live-reading-vns) )
)

(*======================33===============33============333S3========3333:*

(*•*

(*•** (BUG.DEF-VN:READING-VN:CHOOSE-ME VN READING-VN)

**•** Chooses a good location for VN, a DEF, to be read by one of 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 )
  (let ( (nea (constant:constant-generator-nec (vn:name vn) ) )
  (let ( (dest-ne (bug.vn:dest-ne reading-vn) ) )
    (loop (for (ae register) ln (vn:locations vn) )
      (bind delay (ae:ne:delay ae deat-ae) )
      (when (II (== ne dest-ne)
        (< delay best-delay) )
      (do
        (:- best-delay delay)
        (:= best-ae ae) ) )
    )
  )
)

(BUG.DEF-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, hans).

(*•** (defun bug.def-open-constant-vn:reading-vn:choose-me ( vn reading-vn )
  (let ( (best-ne best-cg-ne best-cg-delay)
    (bug.def-constant-vn:reading-vn:best-cg-ne*cycle vn reading-vn) )
  (defun bug.def-closed-constant-vn:reading-vn:choose-me ( vn reading-vn )
    (let ( (best-ne best-cg-ne best-cg-delay)
      (bug.def-constant-vn:reading-vn:best-cg-ne*cycle 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 ( (dest-me (bug.vn:dest-me reading-vn) ) )
    (loop (for (me register) in (vn:locations vn) )
      (bind delay (me:ne:delay me dest-me) )
      (when (II (== me dest-me)
        (< delay best-delay) )
      (do
        (:= best-delay delay)
        (:= best-me me) ) )
    )
  )
)

USES.LSP.57
(defun bug.def-open-var-vn:reading-vn:choose-me (vn reading-vn)
  (bug.def-var-vn:reading-vn:choose-me
   vn reading-vn
   *is.register-bank-mes*)
)

(defun bug.def-closed-var-vn:reading-vn:choose-me (vn reading-vn)
  (if reading-vn
   (vn:likely-mes vn)
   *is.register-bank-mes*)
)

(defun bug.def-var-vn:reading-vn:choose-me (vn reading-vn feasible-mes)
  (list
   (dest-ne
    (bug.vn:dest-ne vn))
   (close-mes
    (loop (for me in feasible-mes)
      (when (fn (me:registers-left me) 0)
        (> (me:registers-left me))))
      (bind delay (me:delay me dest-me))
      (save-minuses me delay)
      )
    )
   )
)

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

(defun bug.vn:me:register-conflicts reading-vn me)
  (if (! reading-vn) (then
    close-mes
  )
  )
  (loop (for me in close-mes) (save-minuses me
    (bug.vn:me:register-conflicts reading-vn me)
  )
  )
  (best-me
    (if (eq dest-me least-conflicting-mes) (then
      dest-me
    )
  )
  )
  (else
    (if-let ((common-mes (intersection 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!"
  )
)

(defun bug.vn:me:delay me dest-me)
  (let*( (delay (me:delay me dest-me))
    )
  )

(defun bug.vn:dest-me (vn)
  (caseq (vn:type vn)
   (operation
    (vn:ae vn))
   (copy
    (bug.vn:dest-me (loop (for reading-vn in (vn:reading-vns vn)
      (naxlnlze reading-vn
      (vn:height reading-vn))))
    )
   )
  )
)

(defun bug.vn:me:register-conflicts reading-vn me)
  (if (! reading-vn) (then
    close-mes
  )
  )
  (loop (for me in close-mes) (save-minuses me
    (bug.vn:me:register-conflicts reading-vn me)
  )
  )
  (best-me
    (if (eq dest-me least-conflicting-mes) (then
      dest-me
    )
  )
  )
  (else
    (if-let ((common-mes (intersection 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!"
  )
)
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 or 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 (vn:likely-me-s vn)))
       (reduce + 1)))
    (if (> required-ports (me:read-ports me))
      1
      0))))
(:= *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:bag  
  list-scheduler:bag-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-ell  
  list-scheduler:simulator  
  list-scheduler:lessex-options  
  list-scheduler:lessex  
)  
)
(:= *build-module-list* (append *build-module-list* *ls.build-module-list*)) )
Build the latest version of the list-scheduling compiler.

(resprop '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)
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 naming to those of GENERATE-CODE).

#LS.ENTRY-VNS=

Lists of all VNs with no predecessors 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 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* 0)

  (:= *ls.exit-vns* 0)

  (:= *ls.new-defs-allowed?* 0)

  (:= *ls.loop-trace?* 0)

  (:= *dag.name:vn* 0)

  (:= *dag.name:write-count* 0)

  (:= *dag.name:locations* 0)

  (:= *dag.name:use-vn* 0)

  (:= *dag.live-before* 0)

  (:= *dag.live-after-constants* 0)

  (:= *dag.loop-start?* 0)

  (:= *dag.use?* 0)

)

(defun dag.mk ( live-before source-record-list live-after )

  (dag.initialize)

  (start-trace)

  (loop (for source-record in source-record-list) (do

    (dag.source-record:create-vn source-record)

  ))

)
(defun dag.source-record:create-vn (source-record)
  (let* ((oper (operator (make-instance 'dag:operation)
          (source-record source-record))
        (group (make-instance 'dag:group)
          (operator oper))
        (trace-direction (trace-direction oper))
        (datum (datum oper))
        (off-live (off-live oper))))
    (cond
      ((= 'def oper)
       (loop (for name in (class oper 'register))
         (do (when (trace name)
               (push 'dag:name:locations name)
               (push 'dag:name:me name)
               (push 'dag:name:register name)))))
      ((= 'use oper)
       (loop (for use-spec in (class oper 'use-spec))
         (do (when (trace use-spec)
               (push 'dag:use? t)
               (push 'dag:use-spec:use-spec use-spec)))))
    (dag.source-record:create-vn oper)))

(procedure dag.source-record:create-pseudo-op (source-record)
  (let* ((oper (make-instance 'dag:operation)
           (source-record source-record))
         (group (make-instance 'dag:group)
           (operator oper))
         (trace-direction (trace-direction oper))
         (datum (datum oper))
         (off-live (off-live oper))))
    (cond
      ((= 'def oper)
       (loop (for name in (class oper 'register))
         (do (when (trace name)
               (push 'dag:name:locations name)
               (push 'dag:name:me name)
               (push 'dag:name:register name)))))
      ((= 'use oper)
       (loop (for use-spec in (class oper 'use-spec))
         (do (when (trace use-spec)
               (push 'dag:use? t)
               (push 'dag:use-spec:use-spec use-spec)))))
    (dag.source-record:create-pseudo-op oper)))
(defun dag.source-record: create-pseudo-op
   (oper trace-direction datum off-live)
   (let ( (vm (vn:create (vn:new
                      type 'pseudo-op
                      oper oper
                      datum datum) ))
        ( predecessors oper trace-direction vn))
      (vn) )
   )

(defun dag.source-record: create-in-parameter
   (oper trace-direction datum off-live)
   (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) ))
           )
      (:= (nag dag.name:vn* written) vn) )
   )

(defun dag.source-record: create-out-parameter
   (oper trace-direction datum off-live)
   (let ( (vm (vn:create (vn:new
                      type (if (assq (oper:operator oper)
                          'value assign vbase constant
                          fconstant) )
                      copy
                      'operation)
                      datun datun
                      name (if (= 'vstore (oper:group oper))
                          (oper:part oper 'written))
                        oper oper) )
      (pred-let (predecessors oper trace-direction vn))
      (:= (vn:operand-vne vn) (dag.oper:operand-vas oper))
      (:= (vn:constraining-vns vn) vns)
      (:= (vn:constraining-delays vn) delays)
      (:= (vn:off-live-vns vn) (loop (for name in off-live)
            (when (if 'trace name )
            (bind off-live-vn (dag.operand:vn name )
                           (save off-live-vn ))
            (:if (vn:name vn) (then
                          (address-vn (dag.operand:vn '(address .read1))
                          (read-vn (dag.operand:vn read1))
                          (store-op (caseq datatype
                                    (float 'ipstore)
                                    (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) ) )
      )
   )

(defun dag.source-record: create-operation
   (oper trace-direction datum off-live)
   (let ( (vm (vn:create (vn:new
                      type (if (assq (oper:operator oper)
                          'assign fassign vbase constant
                          fconstant)) )
                      copy
                      'operation)
                      datun datun
                      name (if (= 'vstore (oper:group oper))
                          (oper:part oper 'written))
                        oper oper) )
      (pred-let (predecessors oper trace-direction vn))
      (:= (vn:operand-vne vn) (dag.oper:operand-vas oper))
      (:= (vn:constraining-vns vn) vns)
      (:= (vn:constraining-delays vn) delays)
      (:= (vn:off-live-vns vn) (loop (for name in off-live)
            (when (if 'trace name )
            (bind off-live-vn (dag.operand:vn name )
                           (save off-live-vn ))
            (:if (vn:name vn) (then
                          (address-vn (dag.operand:vn '(address .read1))
                          (read-vn (dag.operand:vn read1))
                          (store-op (caseq datatype
                                    (float 'ipstore)
                                    (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) ) )
      )
   )

PS:C.S.BULLDOG.LIST-SCHEDULER.TEST>DAG.LSP.70
(defun dag.oper:operand-vns (oper)
  (caseq (oper:operator oper)
    (Upload fpload)
      '((dag.operand:vn (oper:part oper 'index)) )
    (Insert fstore)
      '((dag.operand:vn (oper:part oper 'index))
        (dag.operand:vn (oper:part oper 'value)) )
    (Load fpload)
      '((dag.operand:vn (oper:part oper 'vector))
        (dag.operand:vn (oper:part oper 'index))
        (dag.operand:vn (oper:part oper 'value)) )
    (Value fvaluate)
      '((dag.operand:vn (oper:part oper 'vector))
        (dag.operand:vn (oper:part oper 'index))
        (dag.operand:vn (oper:part oper 'value)) )
    (Load fload)
      '((dag.operand:vn (oper:part oper 'vector))
        (dag.operand:vn (oper:part oper 'index))
        (dag.operand:vn (oper:part oper 'value)) )
    (Assign fassign)
      '((dag.operand:vn (oper:part oper 'vector))
        (dag.operand:vn (oper:part oper 'index))
        (dag.operand:vn (oper:part oper 'value)) )
    (Vbase fbase)
      '((dag.operand:vn (oper:part oper 'vector))
        (dag.operand:vn (oper:part oper 'index))
        (dag.operand:vn (oper:part oper 'value)) )
    (loop (for operand in (oper:part oper 'read))
      (save (dag.operand:vn operand)) )
  )
)

(defun dag.operand:vn (operand)
  (if (litatom operand)
    (vn:create (vn:new type 'def name operand
      locations (dag.name:locations operand)))
    (else (assert (eq operand *dag.live-before*
      "DAG: Name not mentioned in LIVE-BEFORE list."))
      (vn:create (vn:new type 'def
        name operand
        locations (dag.name:locations operand)))
    )
  )
)

(defun dag.constant:create-vn (constant)
  (let ((datatype (caseq (typep constant)
          (double float) 'float)
        (integer) )
    (locations (dag.name:locations constant)
      (immediate? (constant:constant-generator-mes constant))
    )
  )

(defun dag.constant:assign-vn (constant)
  (let ((address-vn (dag.operand:vn '(address .constant))
        (load-op (caseq datatype (float 'fpload)
          (integer 'ipload) )
    )
  )

(defun dag.constant:load-vn (constant)
  (let ((address-vn (dag.operand:vn '(address .constant))
        (load-op (cases datatype float 'fpload)
          (integer 'ipload) )
    )
  )

(defun dag.constant:store-vn (constant)
  (let ((address-vn (dag.operand:vn '(address .constant))
        (load-op (cases datatype float 'fpload)
          (integer 'ipload) )
    )
  )
)

(defun dag.constant:increment-vn (constant)
  (let ((address-vn (dag.operand:vn '(address .constant))
        (load-op (cases datatype float 'fpload)
          (integer 'ipload) )
    )
  )

(defun dag.constant:decrement-vn (constant)
  (let ((address-vn (dag.operand:vn '(address .constant))
        (load-op (cases datatype float 'fpload)
          (integer 'ipload) )
    )
  )
)

(defun dag.constant:store-vn (constant)
  (let ((address-vn (dag.operand:vn '(address .constant))
        (load-op (cases datatype float 'fpload)
          (integer 'ipload) )
    )
  )
)
(defun dag.live-name:create-use (name)
  (if-let ((vn (dag.name:use-vn name)) (then vn))
    (else
     (= (dag.name:use-vn name)
        (vn:create (vn:new
                    type 'use
                    name name
                    operand-vns '((dag.operand:vn name) ))))))
)

(defun dag.use-spec:create-use (name me register)
  (if-let ((vn (dag.name:use-vn name)) (then vn))
    (else
     (= (dag.name:use-vn name)
        (vn:create (vn:new
                    type 'use
                    name me
                    operand-vns '((dag.operand:vn name) ))))))
)

(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))
                (let ( (pred-oper (vn:operand pred))
                           (if (property? pred-oper 'vector-reference)
                               (let ( (part (cdr pred-oper))
                                         (let ( (elt (cdr pred-oper))
                                                  (let ( (type (part pred-oper)))
                                                    (if (== type 'vector-reference)
                                                        (push vns pred)
                                                        (push delays 1)))))
                                    (t (error (list pred-list "DAG: Invalid pred-list."))))
                             (result (reverse vns)
                                    (reverse delays))))))
)

(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))
()
)

(defun dag.name:trace-write-count (name)
  (h »dag.name:write-count » name)
)

(defun dag.name:use-vn (name)
  (h »dag.name:use-vn » name)
)

(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)))))
    ))
)

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

(defun dag.vn:depth-first-print (vn)
  (if (dag.visited-vns* vn)
      (save vn)
    )))
  (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)))))
  )
)

(defun dag.print (optional depth-first?)
  (let ((dag.visited-vns* «set.empty-set»))
    (if depth-first? (then
      (if (vn:is depth-first?) (then
        (dag.vn:depth-first-print depth-first?)
      )
      (else
        (loop (for-each-vn vn) (do
          (vn:print vn))))
    )
    )
  )
)

(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))))
  )
)

(defun dag.print (optional depth-first?)
  (let ((dag.visited-vns* »set.empty-set))
    (if depth-first? (then
      (if (vn:is depth-first?) (then
        (dag.vn:depth-first-print depth-first?)
      )
      (else
        (loop (for-each-vn vn) (do
          (vn:print vn))))
    )
    )
  )
)

(defun dag.name:trace-write-count (name)
  (h »dag.name:write-count » name)
)

(defun dag.name:use-vn (name)
  (h »dag.name:use-vn » name)
)

(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)))))
    ))
)
(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 els.entry-vns) (do
    (assert (&& (= 'def (vn:type vn))
      (vn:operand-vns vn))
    (h vn))
  ) )

  (loop (for vn in els.exit-vns) (do
    (assert (|| (= 'use (vn:type vn))
      (vn:name vn))
    (vn:readng-vns vn)
    (vn:constraining-vns vn))
    (h vn))
  ) )

  (loop (for each-vn vn) (do
    (if (&& (! (vn:operand-vns vn))
      (vn:constraining-vns vn))
      (assert (&& (memq vn els.entry-vns)
        (= 'def (vn:type vn)))
      (h vn))
    )

    (loop (for operand-vn in (vn:operand-vns vn)) (do
      (assert (memq vn (vn:readng-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))
      (assert (&& (memq vn els.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))
    )
  )
)

;***
;*** (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:CONSTRANNING-CYCLE CYCLE PD SD CD)
;***
;***
(defun cycle:delays:constraining-cycle (cycle pd sd cd)
  (- cycle (+ cd (max 0 (- pd sd)))))

13
PS:<C.S.BULLDOG.LIST-SCHEDULER.TEST>DAG.LSP.70

14
(declare (special
    *hash-table.not-found*
    ;**• MACHINE-MODEL
    ;***
    *ls.vn-universe*
    *ls.vn-universe*
    ;*** 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")
  ) )
The ELI model used by Rutt's CS 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 (1 (unique-name unique-fields))
  (let ((cl (atomconcat 'c 1))
        (cl-8 (atomconcat 'c (mod (- 1 8) 8)))
        (cl-1 (atomconcat 'c (mod (- 1 1) 8)))
        (cl+8 (atomconcat 'c (mod (+ 1 8) 8)))
        (cl+1 (atomconcat 'c (mod (+ 1 1) 8)))
        (cl-t-l (atomconcat 'c (mod (- 1 1) 8)))
        (cl+8 (atomconcat 'c (mod (+ 1 8) 8)))
        (cl-t-r (atomconcat 'c (mod (- 1 8) 8))))
    '(
      (def-ne (name (cl r-read))
        (type register-bank)
        (size 64)
        (inputs (cl r-read) (cl r-write))
        (resources (cl r-read) (cl r-write))
        (def-resource-class 'cl r-read)
        (def-resource-class 'cl r-write)
        (def-resource-class resource-name number-of-values))
    )
)

(:= em.floating-test
  '([[=]
      (delay 0)
      (inputs (cl r) (cl c))
      (resources (cl r) (cl c))
      (operators (float fix int div sub eq leq le ge leq lt ge leq))))
  )

; Ce: Register bank CLR

; Ce: Integer tester Cl-

; Ce: Memory CIM

; Ce: Constant generator Clc

; Ce: Unique functional-unit

; Ce: Bus connections

; Ce: Integer adder Cl-
```
(:= em.floating-multiply
  '(f0
   (delay 3)
   (operators (fmul fmul)))))

(:= em.floating-add
  '(f1
   (delay 2)
   (operators (fsub fadd fneg fsin fneg fmax fabs fdiv cos esin sqrt flt fgt
               feq fne fge frecip fquqrt))))

(defun em.immediate-constant? (constant)
  (if (inusp constant)
      (and (< constant 2047)
           (> constant -2048))
      (<= 0 constant)
      (consp constant)
      (eq (car constant) 'addres))
  (t))

(defun load-constant? (constant)
  (if (em.immediate-constant? constant) (then
      (= 'load-em.immediate-constant-action)
    (else t))
    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)
  )
)
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 exercise 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 = 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 DEF's or USE's); 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.

=================================================================================

(defun SAVE-TRACE (name)

(let ( (trace-name (concat name "TR_TRACE")) )

(list ( (new-source-record-list) )

(predecessors-list)

(first-op (operator (car *gch.source-record-list*)))))

(last-op (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*

(append 'def '(use) () ()()))))

(START-TRACE)

(loop (for (oper trace-direction datum off-live) in *gch.source-record-list*)

(inx i from 0)

(do (push new-source-record-list

'(.oper .trace-direction .1 .off-live))

(push predecessors-list

(if (oper:property? oper 'pseudo-op)

())

(predecessors oper trace-direction 1)))

(result

(:= new-source-record-list (dreverse ftftft))

(:= predecessors-list (dreverse ftftft)))))

(fmsg file

#: (trace-name "t t"

"(" t t

"(" t t

(name:datatype name) ""

(name:rank name) ""

")")

))

(t t)

(b *gch.live-before 10000 10000) t t

(b new-source-record-list 10000 10000) t t

(b *gch.live-after 10000 10000) t t

(b predecessors-list 10000 10000) t t

(filename file) ) )

PS: <C.S.BULLDOG.LIST-SCHEDULER.TEST>GENERATE-CODE-HOOK.LSP.5
(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 *la.exit-vns*) (do
 (hd.vn:assign-depth vn)) )
 (loop (for vn in *la.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:constraining-cycle
 (hd.vn:assign-depth constraining-vn)
 (hd.vn:estimated-delay constraining-vn)
 (vn:constraining-vn:delay vn constraining-vn)
 (vn:constraining-delay vn)))
 (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:constrained-delay vn)))
 (vn:height vn)) ) )

(defun hd.vn:estimated-delay ( vn )
 (caseq (vn:type vn)
 (operation
 (+ 1 (operator:min-delay (vn:operator vn)))
 (t
 (error (list vn "HD.VN:ESTIMATED-DELAY: Case error.")))))
)
Top level driver for invoking the list-scheduling compiler.

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

(declare (special
  lsex.code
  lsex.actual-params
  lsex.unoptimized-naddr
  lsex.optimized-naddr
  lsex.compacted-program
  gch.automatic?)

; The most recent TinyLisp code.
; The most recent actual parameters.
; The NADDR straight out of TinyLisp.
; The flow-analysis-optimized NADDR.
; The compacted ELI program.

; 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-mem
  sch.schedule
))

(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) ) t)
)

(unwind-protect
  (let ()
    (if lsex.timed-functions (then
      (apply 'time-functions lsex.timed-functions))
     )
    (:= lsex.compacted-program
      (lis-to-eli (compact lsex.optimized-naddr))
     )
    (msg 0 t)
    (simulate lsex.compacted-program lsex.actual-params)
    (simulator.print-execution-statistics)
    (if lsex.timed-functions (then
      (apply 'time-functions lsex.timed-functions)
     )
     )
    (:= gch.automatic? ()
   )
   )
)

(defun lsex.re-cg ()
  (lsex.initialize t)
  (unwind-protect
    (let ()
      (if lsex.timed-functions (then
        (apply 'time-functions lsex.timed-functions))
       )
      (:= lsex.compacted-program
        (lis-to-eli (compact lsex.optimized-naddr))
       )
      (msg 0 t)
      (simulate lsex.compacted-program lsex.actual-params)
      (simulator.print-execution-statistics)
      (if lsex.timed-functions (then
        (apply 'time-functions lsex.timed-functions)
       )
       )
      (:= gch.automatic? ()
     )
     )
   )
)

(defun lsex.initialize (re-cg?)
  (tr.initialize)
  (initialize-code-generator)
  (if (f) (re-cg?) (then
    (fa.initialize)
    (de.initialize)
    (:= lsex.code ()
     )
    (:= lsex.actual-params ()
     )
    (:= lsex.unoptimized-naddr ()
     )
    (:= lsex.optimized-naddr ()
     )
   )
  )
  (let (old (ggag t))
    (ggag old)
  )
)
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.
RETAIINED-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.
RETAIINED-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.
")
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 ...) (TYPE ...) (DEF-ME ...) (DEF-CONNECTION ...) )

Definea a machine model. Undocumented.

**MA**

(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**

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*

/List-Functional-Unit-MES*

/List-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
  *me.name:me*
  *me.me:me:resources*
  *me.connections*
  *me.operator:me*
  *me.operator:min-delay*
  *me.constant:me*)
)

(eval-when (eval collapse 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:anonymous object number)
        (initial-size 100)))
  )

(defun machine-model.initialize ()
  (resource.initialize)
  (shortest-path.initialize)
  (me-universe:initialize)
  (:= *me.name:me* (hash-table:create () () ()))
  (:= *me.operator:me* (hash-table:create () () ()))
  (:= *me.operator:min-delay* (hash-table:create () () ()))
  (:= *me.me:me:resources* ()))

(defun me:create (me)
  (me-universe:add me)
  (:= (ls.me.name:me* (me:name me)) me)
)

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

  ;* Initialize the register bank MEs. *
  (loop (for each in *ls.register-bank-ae*) (do
    (:= (me:values me) (makevector (me:size me)))
    (:= (me:cycles-free me) (makevector (me:size me)))
    (:= (me:values-names me) (makevector (me:size me)))
    (:= (me:avoids 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 in *ls.register-bank-ae*) (do
    (:= (me:inputs me) (name-list:me-list ???))
    (:= (me:outputs me) (name-list:me-list ???))
  )

  (loop (for each in *la.fundamental-units-ae*) (do
    (error (list each "Case error.") )
  )

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

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

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

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

PS: <C.S.BULLDOG.LIST-SCHEDULER.TEST>MACHINE-MODEL.LSP.54
(loop (for (name1 name2 resource-request) in *mm.connections*)
  (bind me1 (name:se name1)
    me2 (name:se name2))
  (do (assert (eq me2 (me:outputs me1)) )
      (:= (a er *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 (a er *mm.me:me:operators* operator) me)
        (:= (a er *mm.me:me:operators* 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))
)

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

(defnacro def-status-ok? me status
  `(if (me:constraint-function me)
    (funcall (me:constraint-function me) status)
    nil)
)

(defnacro def-operator:min-delay (operator)
  `(a er *mm.me:me:operators* operator)
)

(defnacro def-operator:functional-unit (operator)
  `(a er *mm.me:me:operators* operator)
)

(defnacro def-constant:constant-generator (constant)
  `(let ((list (me (me constant))))
    (if (hash-table:not-found))
      (do (loop for me in *mm.me:me:constant-generator")
        (save me)))
)

(else (me))
)
This module converts 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—preter 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).

(defun mis-to-eli ( mi-list )
  (assert (listp mi-list) )
  (let ( (instr-stream () )
      (instr () )
      (popers () )
      (cond-oper () )
      (trace-dir (mi:trace-direction mi) )
      (sucses (mi:succs mi) )
      (pop-ops (push opers '() label . (ate.al:lnetr-label al) ) )
      (pop-succs)
      (if (opers 'stop) (then
do)
      (caseq operator
        ( () trace copy noop)
        (t (caseq (operator:group operator)
          goto)
          (if-compare if-boolean)
          (push popers
            ('.fu
              .operator
              .operands
              .if-boolean
              .operands
              .if-compare
              (car trace-dir)
              (right)
                .(ate.al:lnetr-label
                  .car succs)
                .(ate.al:lnetr-label
                  .car succs)))))
          (pop succs)
          (pop trace-dir) )
          (t (push popers oper) ))) )))
  (loop (for mi in mi-list) (do
do)
    (push instr-stream instr) )
    (reverse instr-stream) )
  ))
)
(do
  (push *ate.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 (! succ)
  (then
    (assert (! (al:succs al) ) ) ;*** This must be the last MI.
  (else
    (assert (== 1 (length succs) )
      (if (! (al:mi:to-do)
          (== (car succs) (car *ate.mis-to-do*) )
        (then
          (push popers
            '((() goto ,(ate.al:lnetr-label (car succs) ) ) ) ) ) )
        (:= (al:translated-to-source al) (dreverse popers) ) ) )
)

;;;;=====================================================================

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

(defun ate.mi:instr-label ( mi )
  (atomconcat '1 (al:number mi) ) )
* * *

The ELI model used by Rut's CG experiments, modified to have multiple register banks per cluster with a partial crossbar.

(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 ael in me-list1) (splice
            (loop (for ae2 in me-list2) (splice
              ',(def-connection ,ael ,ae2 ,resource-name)))))))))

(defun cluster (1 (unique-name unique-fields))
  (let ((ci (atomconcat 'c 1))
        (ci-3 (atomconcat 'c (mod (- 1 8) 0)))
        (ci-1 (atomconcat 'c (mod (- 1 1) 0)))
        (ci+1 (atomconcat 'c (mod (+ 1 1) 0)))
        (ci-8 (atomconcat 'c (mod (+ 1 8) 0)))
        (ci 'ri-read)
        (ci 'ri-write)
        (ci 'r1)
        (ci 'r2)
        (ci 'r3)
        (ci 'r8)
        (ci 'r9)
        (ci 'r10)
        (ci 'r11)
        (ci 'n')
        (read-ports 2)
        (write-ports 1)
        (read-resources (,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 'r8
                         ,ci 'r9
                         ,ci 'r10
                         ,ci 'r11
                         ,ci 'n')
        (def-resource-class ,resource-name (,ci 'r1-read) 2)
        (def-resource-class ,resource-name (,ci 'r1-write) 1))
  (def-resource-class ,resource-name (,ci-1 'r2-read) 1)
  (def-resource-class ,resource-name (,ci-1 'r2-write) 1)
  (def-resource-class ,resource-name (,ci-8 'r3-read) 2)
  (def-resource-class ,resource-name (,ci-8 'r3-write) 1))

(defun (./* cl )
  (read-ports 2)
  (write-ports 1)
  (read-resources (,ci-3 'r3-read) 2)
  (def-resource-class ,resource-name (,ci-3 'r3-read) 2))

(defun (./* ci )
  (def-resource-class ,resource-name (,ci 'r4-read) 2)
  (def-resource-class ,resource-name (,ci 'r4-write) 1))

(defun (./* ci+) (type functional-unit))

(defun (./* ci'r2)
  (write-ports 1)
  (read-ports 2)
  (read-resources (,ci-3 'r2-read) 2)
  (def-resource-class ,resource-name (,ci-3 'r2-read) 2))

(defun (./* ci'r3)
  (write-ports 1)
  (read-ports 2)
  (read-resources (,ci-3 'r3-read) 2)
  (def-resource-class ,resource-name (,ci-3 'r3-read) 2))

(defun (./* ci'r4)
  (write-ports 1)
  (read-ports 2)
  (read-resources (,ci-3 'r4-read) 2)
  (def-resource-class ,resource-name (,ci-3 'r4-read) 2))

(defun (./* ci'r8)
  (write-ports 1)
  (read-ports 2)
  (read-resources (,ci-3 'r8-read) 2)
  (def-resource-class ,resource-name (,ci-3 'r8-read) 2))

(defun (./* ci'r9)
  (write-ports 1)
  (read-ports 2)
  (read-resources (,ci-3 'r9-read) 2)
  (def-resource-class ,resource-name (,ci-3 'r9-read) 2))

(defun (./* ci'r10)
  (write-ports 1)
  (read-ports 2)
  (read-resources (,ci-3 'r10-read) 2)
  (def-resource-class ,resource-name (,ci-3 'r10-read) 2))

(defun (./* ci'r11)
  (write-ports 1)
  (read-ports 2)
  (read-resources (,ci-3 'r11-read) 2)
  (def-resource-class ,resource-name (,ci-3 'r11-read) 2))

(defun (./* ci'n)
  (write-ports 1)
  (read-ports 2)
  (read-resources (,ci-3 'n-read) 2)
  (def-resource-class ,resource-name (,ci-3 'n-read) 2))
REGISTERS

This module handles the allocation and deallocation 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 occupying 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. An error is raised if there are no free registers or if REQUIRED-REGISTER is specified but not free.

When choosing 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 guarantee that the register will be free).

Then we look in the preferred locations of the name of VN 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 occupying 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 re-compute 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 ae ln *register-bank-aes*) (do
    (:= (ne:registers-left ae) (ne:size ae)
    (vector:initialize (ne:value ae) ()
    (vector:initialize (ne:cycle-free ae) 0)
    (vector:initialize (ne:value-names ae) ()
    (vector:initialize (ne:avoid? ae) ()
    )
  )
  )

(defun cycle:me:register:occupied? ( cycle me register )
  (if (> (me:registers-free me) 0) (then
    (me:registers-free me)
  ) (else
    ()
  ))

(defun cycle:me:vn:free-register? ( cycle me vn )
  (if (> (me:registers-free me) 0) (then
    (me:registers-free me)
  ) (else
    :*** Recompute the :REGISTERS-LEFT for this cycle. See notes
    *** above.

    (:= (me:registers-free me)
    (loop (incr 1 fron 0 to (- (me:size me) 1)
    (when (> cycle (me:cycles-free me) 1)
      (reduce + 0 1)
    )
    (me:registers-free me)
    )
  ))

(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

(defun vn:record-preferred-location (vn)
  (assert (vn:register vn))
  (let (ne (vn:reglater-bank-ne vn)
         register (vn:reglater-ne vn))
    (:= (if (me:avoid-ne register) t)
        (push ((vn:record-name locations* (vn:name vn)) 
        (.me .register)))))

(let (me (vn:register-bank-ne vn))
  (register (vn:register vn))
  (:= (if (me:avoid-ne register) t)
       (push ((vn:record-name locations* (vn:name vn)) 
       (.me .register)))))
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 otherwise. If false, the schedule is left in a garbage 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-TRANSACTION)

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-TRANSACTION 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.

(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-size-class*
  *res.cycle:resource-set*
  *res.transaction-in-progress*
  *res.transaction-size*
  *res.saved-bytes*
  *res.saved-positions*
  *res.saved-cycles*
  *res.failure-type*)

(defn resource.initialize ()
  (= *res.all-resource-classes* ()
    (= *res.total-classes* 0)
    (= *res.max-size-class* 0))
(:= *res.name:resource-class* (hash-table:create () () ()))
(:= *res.cycle:resource-set* () )
(:= *res.transaction-in-progress?* () )
(:= *res.transaction-size* 0)
(:= *res.saved-byte* (makevector 0))
(:= *res.saved-position* (makevector 0))
(:= *res.saved-cycles* (makevector 0))
(:= *res.failure-type* (ls))

(defun resource.initialize-schedule ( failure-type )
(:= *res.cycle:resource-set* (makevector *res.transaction-size*))
(let ( (byte-size (ceiling (log 2) ) )
(loop (incr 1 from 0 to (- *res.transaction-size* 1))
(do (:= ([] *res.cycle:resource-set* 1)
(byte-vector:create *res.total-classes* byte-size))))
(:= *res.transaction-in-progress?* () )
(:= *res.transaction-size* 0)
(:= *res.saved-byte* (makevector 100)) ;*** Big enough?
(:= *res.saved-position* (makevector 100)) ;*** Big enough?
(:= *res.saved-cycles* (makevector 100)) ;*** Big enough?
(assert (eq failure-type '(ls bug)) )
(:= *res.failure-type* failure-type))

(defun resource-class:create ( name size )
(let ( (resource-class:new name size position
ls-failures 0 bug-failures 0 ) )
(:= *res.total-classes* (+ & & 1))
(:= *res.max-class-size* (max & & size))
(:= *res.all-resource-classes* (appendl & & resource-class))
(:= ([] h res.name:resource-class* name) 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 item in cycle-request) (save
(if (cons item) 
(('(name:resource-class item) ,(cadr item)) ) ) ) ) )

(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 1 from (- *res.transaction-size* 1) to 0) (do
(:= ([] *res.cycle:resource-set* 1)
(byte-vector:create *res.saved-byte* 1) )
(:= *res.saved-byte* 1) ) )
(:= *res.transaction-in-progress?* () )
(:= *res.transaction-size* 0)
)

(defun cycle:resource-request:available? ( cycle resource-request )
(loop (incr 1 from cycle)
(for cycle-request in resource-request)
(when (I (resource-set:cycle-request:available?
([] res.cycle:resource-set* 1)
(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?
(res.cycle:resource-set* 1)
(cycle-request)) )
(result cycle) )

(defun cycle:resource-request:schedule? ( cycle resource-request )
(loop (incr 1 from cycle)
(for cycle-request in resource-request)
(when (I (cycle:cycle-request:schedule? 1)
(cycle-request)) )
(do (return () )
(result t) )
)

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

PS:<C.S.BULLDOG.LIST-SCHEDULER.TEST>RESOURCE.LSP.41
(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)
            (resource-class:position resource-class)))
      (do
        (if (= 'ls resource-class:failure-type*) (then
          (resource-class:ls-failures resource-class) (+ 1))
        (else
          (resource-class:bug-failures resource-class) (+ 1)))
      (return 1))
    (result t))
  (defun cycle:cycle-request:schedule? (cycle cycle-request)
    (let ( (resource-set (resource-set (resource-class:position resource-class)) )
      (loop (for (resource-class request-size) in cycle-request)
        (bind old-resources
          (resource-class:position resource-class)
          new-resources
          (+ request-size old-resources)
        (do
          (if (> new-resources
              (resource-class:size resource-class))
            (then
              (if (= 'ls resource-class:failure-type*) (then
                (resource-class:ls-failures resource-class) (+ 1))
              (else
                (resource-class:bug-failures resource-class) (+ 1)))
              (return 1))
            (else
              (resource-set (resource-class:position resource-class)
                new-resource)
              (result t))
      (defun resource-set:equal (resource-set1 resource-set2)
        (loop (incr 1 from 0 to (+ *res.total-classes* 1))
          (when (!== (resource-set1 1)
                      (resource-set2 1))
            (do
              (return 1))
          (result t))
    (defun resource-set:print (resource-set)
      (let (first-set)
        (loop (for resource-class in *res.all-resource-classes*
          (bind resources
            (resource-class:position resource-class)
            (* (resource-class:name resource-class) resources))
          (splice
            (? (ss o resources)
              0)
            (ss i (resource-class:else resource-class)
              (assert (ss i resources))
              (.(resource-class:naae resource-class))
            t
              (* (.(resource-class:naae resource-class) resources)))))))
  (defun resource:print-schedule ()
    (msg 0)
    (loop (initial first-1 0)
      (while (< first-1 *ls.max-schedule-sizes*
        (bind first-set (resource-set resource-set first-1))
        (loop (incr last-1 from (+ first-1 1) to (- *ls.max-schedule-sizes* 1))
          (while (resource-set:equal first-set
            (resource-set:position resource-class))
            (result
              (= last-1 (- last-1 1))
              (if (> last-1 first-1)
                (msg (j first-1 8) (*: (j last-1 8) ""))
                (result t))
        (msg 0 t "Resource failures:" (t 20) (je "LS" 5) (t 80) (je "BUS" 6) t)
        (loop (for resource-class in sort *res.all-resource-classes*
          (do
            (msg (resource-class:name resource-class) t 20)
            (t 20) (je (resource-class:ls-failures resource-class) 6)
            (t 80) (je (resource-class:bug-failures resource-class) 5) t))
  (defun resource:print-failures ()
    (msg 0 t "Resource failures:" (t 20) (je "LS" 5) (t 80) (je "BUS" 6) t)
    (loop (for resource-class in sort *res.all-resource-classes*
      (do
        (msg (resource-class:name resource-class) t 20)
        (t 20) (je (resource-class:ls-failures resource-class) 6)
        (t 80) (je (resource-class:bug-failures resource-class) 5) t))
  (defun resource:print-schedule ()
    (msg 0)
    (loop (initial first-1 0)
      (while (< first-1 *ls.max-schedule-sizes*)
        (bind first-set (resource-set resource-set first-1))
        (do
          (loop (incr last-1 from (+ first-1 1) to (- *ls.max-schedule-sizes* 1))
            (while (resource-set:equal first-set
              (resource-set:position resource-class))
              (result
                (= last-1 (- last-1 1))
                (if (> last-1 first-1)
                  (msg (j first-1 8) (*: (j last-1 8) ""))
                  (result t))
          (msg 0 t "Resource failures:" (t 20) (je "LS" 5) (t 80) (je "BUS" 6) t)
          (loop (for resource-class in sort *res.all-resource-classes*
            (do
              (msg (resource-class:name resource-class) t 20)
              (t 20) (je (resource-class:ls-failures resource-class) 6)
              (t 80) (je (resource-class:bug-failures resource-class) 5) t))
        (msg 0)
        (loop (initial first-1 0)
          (while (< first-1 *ls.max-schedule-sizes*)
            (bind first-set (resource-set resource-set first-1))
            (do
              (loop (incr last-1 from (+ first-1 1) to (- *ls.max-schedule-sizes* 1))
                (while (resource-set:equal first-set
                  (resource-set:position resource-class))
                  (result
                    (= last-1 (- last-1 1))
                    (if (> last-1 first-1)
                      (msg (j first-1 8) (*: (j last-1 8) ""))
                      (result t))
              (msg 0))
            (msg 0 t "Resource failures:" (t 20) (je "LS" 5) (t 80) (je "BUS" 6) t)
            (loop (for resource-class in sort *res.all-resource-classes*
              (do
                (msg (resource-class:name resource-class) t 20)
                (t 20) (je (resource-class:ls-failures resource-class) 6)
                (t 80) (je (resource-class:bug-failures resource-class) 5) t))
        (msg 0))
(eval-when (compile load)
 (include list-scheduler:declarations) )

(*SCH.COPY-VN:SCHEDULE VN*)

(*SCH.COPY-VN:SCAPE VN*)

(ach.copy-vn:schedule vn)

(defun ach.copy-vn:schedule? vn)

(defun sch.copy-vn:schedule vn)

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

;*** Mark the operand as having been read this cycle.
;***
(= (vn:read-cycle operand-vn) 0)

;*** Assign the register location to this VN, inserting
;*** copy VNs as necessary, 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
 (= 1s.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.vn:copy-vn:schedule-machine-operation vn)

(return 'success) ) )
(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 ([] *la.schedule* »Is.cycle*)
            (vn:register vn)
            (vn:naae vn)
            (vn:name vn)
            (if (constant-generator (me:type source-me))
                (vn:name operand-vn)
                (vn:register operand-vn)
                (vn:name operand-vn)) )
    (:= «la.last-cycle* (max & & & «ls.cycle*) ) )
)

(SCH.COPY-VN:BLOCKING-VN:MAKE-DEPENDENT VN BLOCKING-VN)
*** VN is a COPY that, due to a USE, 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
*** reader, 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 requested for the next cycle (usually it shouldn't — see
*** below).
***
(defun sch.copy-vn:blocking-vn:make-dependent (vn blocking-vn)
  (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:schedule-reading-vns)
    (vn:readers-left blocking-vn)
    (+ 1 (v: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:constraining-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 )
  (t ( (vn:scheduled-cycle descendant-vn)
  (t)
  (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) ) ) ) )

PS:<C.S.BULLDOG.LIST-SCHEDULER.TEST>SCH-COPY.LSP.68
Attenpts to schedule VN, an operation, requeuelng it lf It can't be acheduled for any reason.

If the operation has no readers left, that means Its only use vas an off-live use at a split which has already been scheduled above the operation; the operation ls now useless and can be acheduled as a NOOP.

(defun sch.operation-vn: schedule ( vn )
  (lf (ftft (=s o (vn:readers-left vn) )
    (vn:name vn) )
    (then
      (ach.noop-vn:achedule vn) )
    (elae
      (resource.start-transaction)
      (lf (sch.operatlon-vn:schedule? vn) (then
        (resource.commit-transaction) )
      (else
        (resource.abort-transaction)
        (sch.vn:cycle:enqueue vn (+ *ls.cycle* 1) ) ) )
    )
  )
)

(defun sch.operation-vn: scheduled? ( vn ) (prog ()
  (let ( (fu-ae (vn:ne vn) )
    (unlque-operand-vna (nodupleaq (vn:operand-vns vn) )
      (dest-ne () )
    (deat-register () )
  )
    ;*** Try scheduling the functional unit resources.
    ;***
    (if (! (cycle:reaource-requeat:achedule? *ls.cycle* (ae:reeourcea fu-ae)))
      then
        (return () )
    (else
      (sch.vn:cycle:enqueue vn (+ *ls.cycle* 1) )
    )
  )
)

(sch.vn:release-operands vn)
(sch.vn:release-cycles 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 ) )

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-ne:schedule-read? (vn fu-me) (prog ()
  (let ((ne (ach.vn:location-ne vn)))
    (if (&= (fnf cycle (vn:read-cycle vn)) (fnf cycle (资源 request: schedule? (不 constant-generator (ne:type ne)) (me:resources me) (me:read-resources me))))
      (return t))
    (return t))
  (return 0)
)

(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 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 (+ els.cycle
      (if (fnf 'functional-unit (me:type source-me)) (me:delay source-me)))))

(sch.operation-vn:schedule-machine-operation vn)

Places the machine operation for VN, an operation, on the schedule.
(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* 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))
))
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 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 in is the set of live VNs on entry to the partial schedule that will be generated to precede 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 be 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 cycle 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:

\[
\text{LIVE}[\text{I}] = (\text{LIVE}[\text{I-1}] - \text{KILL}[\text{I-1}]) \cup \text{GEN}[\text{I}]
\]

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:

\[
\text{LIVE}[\text{I}] = (\text{LIVE}[\text{I-1}] - \text{KILL}[\text{I-1}]) \cup \text{GEN}[\text{I}]
\]

The LIVE vector is used to compute the proper DEFs and USEs at splits and joins. For each cycle, we record the set of VNs that are live on entry to that cycle (whose extents include that cycle).

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 cycle 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:

\[
\text{LIVE}[\text{I}] = (\text{LIVE}[\text{I-1}] - \text{KILL}[\text{I-1}]) \cup \text{GEN}[\text{I}]
\]

The LIVE vector is used to compute the proper DEFs and USEs at splits and joins. For each cycle, we record the set of VNs that are live on entry to that cycle (whose extents include that cycle).

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 cycle 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:

\[
\text{LIVE}[\text{I}] = (\text{LIVE}[\text{I-1}] - \text{KILL}[\text{I-1}]) \cup \text{GEN}[\text{I}]
\]

The LIVE vector is used to compute the proper DEFs and USEs at splits and joins. For each cycle, we record the set of VNs that are live on entry to that cycle (whose extents include that cycle).

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 cycle 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:

\[
\text{LIVE}[\text{I}] = (\text{LIVE}[\text{I-1}] - \text{KILL}[\text{I-1}]) \cup \text{GEN}[\text{I}]
\]

The LIVE vector is used to compute the proper DEFs and USEs at splits and joins. For each cycle, we record the set of VNs that are live on entry to that cycle (whose extents include that cycle).

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 cycle 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:

\[
\text{LIVE}[\text{I}] = (\text{LIVE}[\text{I-1}] - \text{KILL}[\text{I-1}]) \cup \text{GEN}[\text{I}]
\]

The LIVE vector is used to compute the proper DEFs and USEs at splits and joins. For each cycle, we record the set of VNs that are live on entry to that cycle (whose extents include that cycle).

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 cycle 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:

\[
\text{LIVE}[\text{I}] = (\text{LIVE}[\text{I-1}] - \text{KILL}[\text{I-1}]) \cup \text{GEN}[\text{I}]
\]
::: Compute LIVE from GEN and KILL

```lisp
(:= ([ live 0 ] [ gen 0 ]) 
(loop (incr i from 1 to (+ 2 *ls:last-cycle* )) (do 
(:= ([ live i ] )  
(vn-set:union 
(vn-set:difference ([ live (- 1 i ] )  
([ gen i ] ) ) ) ) ) )
```

::: 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.

```lisp
(defun sch.vn:spllt-gen-cycle ( vn )  
(let* ( (cycle 
(caseq (vn:type vn) 
(defl 0)  
(operation (+ (vn:scheduled-cycle vn) 1 ) )  
(copy (+ (vn:scheduled-cycle vn) 1 ) )  
(t () ) ) )
)
```

```lisp
(defun sch.vn:join-gen-cycle ( vn )  
(let* ( (cycle 
(caseq (vn:type vn) 
(defl 0)  
(operation (+ (vn:scheduled-cycle vn) 1 ) )  
(copy (+ (vn:scheduled-cycle vn) 1 ) )  
(t () ) ) )
)
```

::: Hairiness here: If a VN has no readers but just off-live

```lisp
(defun sch.vn:spllt-klll-cycle ( vn )  
(let* ( (cycle 
(caseq (vn:type vn) 
(defl 0)  
(operation (+ (vn:scheduled-cycle vn) 1 ) )  
(copy (+ (vn:scheduled-cycle vn) 1 ) )  
(t () ) ) )
)
```

```lisp
(defun sch.vn:join-klll-cycle ( vn )  
(let* ( (cycle 
(caseq (vn:type vn) 
(defl 0)  
(operation (+ (vn:scheduled-cycle vn) 1 ) )  
(copy (+ (vn:scheduled-cycle vn) 1 ) )  
(t () ) ) )
)
```

::: Constructs a DEF for a split at CYCLE (the last cycle of the jump
::: is scheduled for CYCLE).

```lisp
(defun sch.cycle:spllt-def ( cycle )  
"(def 
",(if *is.trace-information?*  
"'(strace ,*str.trace-number* ,*cycle* ,(+ cycle 1 ) ) )  
",(loop (for-each-vn-set-element ([ ] sch.cycle:spllt-live*  
"(vn)  
"(save *",(vn:name vn)  
"(+ cycle 1 ) ) )  
```

::: 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).

```lisp
(defun sch.vn:spllt-klll-cycle ( vn )  
(let* ( (cycle 
(caseq (vn:type vn) 
(defl 0)  
(operation (+ (vn:scheduled-cycle vn) 1 ) )  
(copy (+ (vn:scheduled-cycle vn) 1 ) )  
(t () ) ) )
)
```

:::Hairiness here: If a VN has no readers but just off-live
(defun sch.cycle:join-use (cycle)
' (use
... (if *ls.trace-information?*
"((\trace, str.trace-number, \cycle, (+ cycle i))")
... (loop (for-each-vn-set-element (\sch.cycle:join-live* cycle) vn)
    (save "+(\name vn))
    ,(ae.name (vn:register-bank-me vn))
    ,(vn:register vn)))))

(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 i) to 0)
    (bind partial-cycle
     (loop (for vn&oper in (\ls.schedule* cycle))
       (bind (vn : oper) vn&oper)
       (when (\= (vn:type vn) )
         (<= (vn:scheduled-cycle vn) split-cycle )))
     (save vn&oper) )
    (while partial-cycle)
    (save
     (if *ls.trace-information?* (then
       '((() . () trace, str.trace-number, split, (+ 1 split-cycle)) )
     (else
      partial-cycle ))))))

(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 i) to *ls.max-schedule-size*)
    (bind partial-cycle
     (loop (for vn&oper in (\ls.schedule* cycle))
       (bind (vn : oper) vn&oper)
       (when (\= (vn:type vn) )
         (<= (vn:scheduled-cycle vn) split-cycle )))
     (save vn&oper) )
    (while partial-cycle)
    (save
     (if *ls.trace-information?* (then
       '(((() . () trace, str.trace-number, split, (+ 1 split-cycle)) )
     (else
      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.
LIST SCHEDULER

This module implements the main part of the list scheduler and its interface to the outside world:

(INITIALIZE-CODE-GENERATOR)
(SETTLE-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 DOCUMENTATION: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))

(defun initialize-code-generator ()
  (vn:initialize)
  (dag:initialize)
  (ach:initialize))

(defun generate-code (live-before source-record-list live-after)
  (vn:initialize)
  (dag:make live-before source-record-list live-after)
  (bug:assign-nes)
  (msg 0 "Last BUG cycle = "
    (loop (for-each-vn vn)
      (when (vn:bug-cycle vn)
        (maxsize
          (+ (vn:bug-cycle vn)
            (caseq (vn:type vn)
              (operation (vn:delay (vn:se vn)) 0) 0)))
    )
  (ach:initialize)
  (ach:initialize)

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

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

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

(defun schedule:[] (schedule 1)
  (let* (vn:opter-list
    (if (= schedule *ls.schedule*)
      ([] *ls.schedule* (- i 1))
      (nth-els schedule i)))
    (result
      (loop (for (vn . oper) in vn:opter-list) (save
        ".oper .vn (vn:datum vn) .vn .vn")
      (if (& & *ls.trace-information"
        (result
          (if (and *ls.trace-information"
            
        (result
          (result)))))
      (else
        (result))))
    (result)
    )

(defun schedule:join (schedule 1)
  (let ((cycle (- i 1)))
    (BCH:cycle:join-use cycle)
    (BCH:cycle:join-partial-schedule cycle)

(defun schedule:split (schedule 1 jump-number)
  (let ((cycle (- i 1)))
    (BCH:cycle:split-def cycle)
    (BCH:cycle:split-partial-schedule cycle)))

;*** SCH.SCHEDULE

;*** This procedure 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.

PS:<C.S.BULLDOG.LIST-SCHEDULER.TEST>SCHEDULER.LSP.163
• sch.vn-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 ()
  (is.schedule 0)
  (la.last-cycle 0)
  (la.cycle 0)
  (ach.data-ready-heap (heap:create 'ach.vn:vn:heap-compare))
  (sch.data-ready-heap (heap:create 'ach.vn:vn:heap-compare))
  (sch.vn-to-release-off-live ()
    (sch.vn8-to-release-off-live)
    (sch.split-live 0)
    (sch.join-live 0)
  )
)

(defun sch.schedule ()
  (sch.initialize)
  (sch.vn:schedule vn)
  (s (s sch.vn:schedule vn))
  (sch.vn:schedule vn)
)

(defun sch.schedule ()
  (sch.initialize)
  (sch.vn:schedule vn)
  (s (s sch.vn:schedule vn))
  (sch.vn:schedule vn)
)

(defun ach.schedule ()
  (sch.initialize)
  (sch.vn:schedule vn)
  (s (s sch.vn:schedule vn))
  (sch.vn:schedule vn)
)

(defun ach.schedule ()
  (sch.initialize)
  (sch.vn:schedule vn)
  (s (s sch.vn:schedule vn))
  (sch.vn:schedule vn)
)

(defun ach.schedule ()
  (sch.initialize)
  (sch.vn:schedule vn)
  (s (s sch.vn:schedule vn))
  (sch.vn:schedule vn)
)

(defun ach.schedule ()
  (sch.initialize)
  (sch.vn:schedule vn)
  (s (s sch.vn:schedule vn))
  (sch.vn:schedule vn)
)

(defun ach.schedule ()
  (sch.initialize)
  (sch.vn:schedule vn)
  (s (s sch.vn:schedule vn))
  (sch.vn:schedule vn)
)

(defun ach.schedule ()
  (sch.initialize)
  (sch.vn:schedule vn)
  (s (s sch.vn:schedule vn))
  (sch.vn:schedule vn)
)

(defun ach.schedule ()
  (sch.initialize)
  (sch.vn:schedule vn)
  (s (s sch.vn:schedule vn))
  (sch.vn:schedule vn)
)

(defun ach.schedule ()
  (sch.initialize)
  (sch.vn:schedule vn)
  (s (s sch.vn:schedule vn))
  (sch.vn:schedule vn)
)

(defun ach.schedule ()
  (sch.initialize)
  (sch.vn:schedule vn)
  (s (s sch.vn:schedule vn))
  (sch.vn:schedule vn)
)

(defun ach.schedule ()
  (sch.initialize)
  (sch.vn:schedule vn)
  (s (s sch.vn:schedule vn))
  (sch.vn:schedule vn)
)

(defun ach.schedule ()
  (sch.initialize)
  (sch.vn:schedule vn)
  (s (s sch.vn:schedule vn))
  (sch.vn:schedule vn)
)

(defun ach.schedule ()
  (sch.initialize)
  (sch.vn:schedule vn)
  (s (s sch.vn:schedule vn))
  (sch.vn:schedule vn)
)

(defun ach.schedule ()
  (sch.initialize)
  (sch.vn:schedule vn)
  (s (s sch.vn:schedule vn))
  (sch.vn:schedule vn)
)

(defun ach.schedule ()
  (sch.initialize)
  (sch.vn:schedule vn)
  (s (s sch.vn:schedule vn))
  (sch.vn:schedule vn)
)

(defun ach.schedule ()
  (sch.initialize)
  (sch.vn:schedule vn)
  (s (s sch.vn:schedule vn))
  (sch.vn:schedule vn)
)
*** Make sure all the VNs were scheduled.

(loop (for-each-vn vn)
  (when (not (equal (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.)
      (do
        (for-some (reading-vn in (vn:reading-vns vn))
          (assert (not (equal (vn:type reading-vn) '(use)))
            (h vn) t "Register *i* in bank *(me:name me)* containing: *t"
            (when vn)
              (do
                (assert (not (equal (vn:readings vn) (.vn:readings vn))
                  (h vn) t "w asn'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)
  (? (le (vn:cycle vn) (vn:cycle vn))
    (? (ge (vn:height vn) (vn:height vn))
      (? (le (vn:bug-cycle vn) (vn:bug-cycle vn))
        (vn:cycle vn) cycle)
      (error (list vn "Case error.")
        ()
      )
    )
  )
)

(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 (le (vn:predecessors-left vn) 0)
    (do
      (heap:remove *sch.data-ready-heap* vn)
      ()
    )
  )
)

(SCH.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

(defun sch.vn:heap:compare ( vn1 vn2)
  (? (le (vn:cycle vn1) (vn:cycle vn2))
    (? (le (vn:height vn1) (vn:height vn2))
      (? (le (vn:bug-cycle vn1) (vn:bug-cycle vn2))
        (vn:cycle vn1) (vn:cycle vn2)
      )
    )
  )
)

PS:<C.S.BULLDOG.LIST-SCHEDULER.TEST>SCHEDULER.LSP.163
(defun sch.def-vn:schedule (vn)
  (let ((dest-me (vn:me vn))
        (dest-register ()))
    (let ((vn:scheduled-cycle vn) (vn:cycle))
      (if (ss 'constant-generator (me:type dest-me))
          (sch.vn:splice-copies vn)
        (else
dest-register))
      (vn:record-preferred-location usei-vn)))))))
(sch.vn:release-successors vn)
)

(sch.vn:release-operands vn)
(sch.vn:release-successors vn)
(sch.vn:release-off-live vn)
(push ([] 'vn 0 'noop)

Schedules a DEF VN merely by scheduling all of its readers which are
DEFI VNs.

VN is a DEF. 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 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 USEIs 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).

VN is either a COPY or an OPERATION which we've decided shouldn't
produce any machine operations. 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
a NOOP that doesn't consume any resources.

Note: In the 2nd edition of BullDogs, the bookkeeper has changed its
interface, and the bookkeeper no longer has a function that
provides a representative for any source operation on the schedule.

In the 2nd edition of BullDogs, it has been assumed that
all source operations are scheduled as NOOPs. This change in
the bookkeeper interface has led to the need to schedule NOOPs.

In the 2nd edition of BullDogs, it has been assumed that
all source operations are scheduled as NOOPs. This change in
the bookkeeper interface has led to the need to schedule NOOPs.

In the 2nd edition of BullDogs, it has been assumed that
all source operations are scheduled as NOOPs. This change in
the bookkeeper interface has led to the need to schedule NOOPs.

In the 2nd edition of BullDogs, it has been assumed that
all source operations are scheduled as NOOPs. This change in
the bookkeeper interface has led to the need to schedule NOOPs.

In the 2nd edition of BullDogs, it has been assumed that
all source operations are scheduled as NOOPs. This change in
the bookkeeper interface has led to the need to schedule NOOPs.

In the 2nd edition of BullDogs, it has been assumed that
all source operations are scheduled as NOOPs. This change in
the bookkeeper interface has led to the need to schedule NOOPs.

In the 2nd edition of BullDogs, it has been assumed that
all source operations are scheduled as NOOPs. This change in
the bookkeeper interface has led to the need to schedule NOOPs.

In the 2nd edition of BullDogs, it has been assumed that
all source operations are scheduled as NOOPs. This change in
the bookkeeper interface has led to the need to schedule NOOPs.

In the 2nd edition of BullDogs, it has been assumed that
all source operations are scheduled as NOOPs. This change in
the bookkeeper interface has led to the need to schedule NOOPs.

In the 2nd edition of BullDogs, it has been assumed that
all source operations are scheduled as NOOPs. This change in
the bookkeeper interface has led to the need to schedule NOOPs.

In the 2nd edition of BullDogs, it has been assumed that
all source operations are scheduled as NOOPs. This change in
the bookkeeper interface has led to the need to schedule NOOPs.

In the 2nd edition of BullDogs, it has been assumed that
all source operations are scheduled as NOOPs. This change in
the bookkeeper interface has led to the need to schedule NOOPs.

In the 2nd edition of BullDogs, it has been assumed that
all source operations are scheduled as NOOPs. This change in
the bookkeeper interface has led to the need to schedule NOOPs.

In the 2nd edition of BullDogs, it has been assumed that
all source operations are scheduled as NOOPs. This change in
the bookkeeper interface has led to the need to schedule NOOPs.

In the 2nd edition of BullDogs, it has been assumed that
all source operations are scheduled as NOOPs. This change in
the bookkeeper interface has led to the need to schedule NOOPs.
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 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) )
  ) )

SCH.VN:RELEASE-SUCCESSORS enqueues the predecessors 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:read-release ( vn )
  (:= (vn:readers-left vn) (- ftftft 1) )
  (if (>s o (:s (vn:predecessors-left vn) (- ftftft 1) ) )
    (then
      (sch.vn:cycle:enqueue reading-vn (vn:cycle reading-vn) ) )
  )
)

SCH.VN:RELEASE-OPERANDS VN
SCH.VN:RELEASE-OFF-LIVE VN
SCH.VN:RELEASE-SUCCESSORS VN
SCH.VN:READ-RELEASE VN
(if (> 0 (:= (vn:predecessors-left constrained-vn) (- ftftft 1) ) )
  (sch.vn:cycle:enqueue constrained-vn
  (vn:cycle constrained-vn))))
)

(defun sch.vn:me:register:cycle:assign-location vn me register cycle )
(:= (vn:register-b_nk-ae vn) Be)
(:s (vn:register vn) reglater)
(:= (vn:register-cycle vn) cycle)
sen.vn:splice-copies vn)
0 )

(defun sch.vn:apHce-coplea ( vn )
(let ( (flrat-on-pathftreadlng-vna-table () ) )
(loop (for readlng-vn ln (vn:readlng-vns vn) )
  (bind flrat-on-path (ach.vn:readlng-vn:needa-copy? vn readlng-vn) )
  (when flret-on-path)
    (do
      (lf-let ( (neftreadlng-vns (assoc flrst-on-path
        flrst-on-pathftreadlng-vns-table)))
        (then
          (push (cdr neftreadlng-vns) readlng-vn) )
        (else
          (push flrst-on-pathftreadlng-vns-table
            '(first-on-path ,readlng-vn) ) ) ) )
  )
(loop (for (() . readlng-vns) in f1rst-on-pathftrending-vns-table)
  (bind copy-vn (vn:create (vn:new
    type 'copy
    (vn:register vn) (vn:register readlng-vn)
    (vn:readere-left vn) ) )
  )
  )
)

(defun sch.vn:readlng-vn:needs-copy? ( vn readlng-vn )
(let ( (source-ae (ach.vn:location-ne vn) ) )
  (? ;*** READING-VN la an operation. A COPY la needed if the
  ;*»• location of VN ls 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.
  ;*••
  (ftft (== 'operation (vn:type readlng-vn) )
    (II (! (nenq eourca-ne (ne:lnputa (vn:ne readlng-vn) ) )
      (ech.vn:ne :readlng-vn: conflict?
        vn source-ae readlng-vn) ) )
    (ne:ne:flret-on-path eource-ne (vn:ne readlng-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 ( (assoc first-on-path
          (vn:readlng-vns vn) )
          (sch.vn:reading-vn:needs-copy?)
          vn reading-vn) )
          (then
            (push (cdr neftreading-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
    (vn:register vn) (vn:register reading-vn)
    (vn:readere-left vn) ) )
  )
  ))

SCH.VN:READING-VN:NEEDS-COPY? VN READING-VN

Returns -() if READING-VN can't read the value of VN from its current
location, false o.w. The -() value returned is usually the first ME
on the path from the location of VN to wherever READING-VN wants it.

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.

SCH.VN:ME:REGISTER:CYCLE:ASSIGN-LOCATION VN ME REGISTER CYCLE

SCH.VN:ME:REGISTER:CYCLE:ASSIGN-LOCATION VN ME REGISTER CYCLE

SCH.VN:ME:REGISTER:CYCLE:ASSIGN-LOCATION VN ME REGISTER CYCLE

SCH.VN:ME:REGISTER:CYCLE:ASSIGN-LOCATION VN ME REGISTER CYCLE

SCH.VN:ME:REGISTER:CYCLE:ASSIGN-LOCATION VN ME REGISTER CYCLE

...
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.

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.

Returns 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 assigned to the VN.
- COPY: wherever the max height reader of VN wants its value.
- USEI: 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 USEI and doesn’t have a required location.

GROAN: We use an unexported function from BUG.

Returns the reader of VN of a max height.

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 DEF assigned a constant generator, in which case the c.g. ME is returned.

Compiler bug.

Returns the reader of VN of a max height.

Returns the reader of VN of a max height.
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-mach (name r)
    (type register-bank)
    (size 8 (* 8 84))
    (inputs (r fa fu3 fu6 r c))
    (read-ports 8)
    (write-ports 1)
    (read-resources (r-read))
    (write-resources (r-write))
  )
  (def-resource-class r-read 8)
  (def-resource-class r-write 1)
  ;*** Constant generator C
  ;***
  (def-mach (name c)
    (type constant-generator)
    (resources (constant))
    (constraint-function sm.immediate-constant?)
  )
  (def-resource-class constant 1)
  ;*** 1-cycle functional unit FUI
  ;***
  (def-mach (name fu1)
    (type functional-unit)
    (delay 0)
    (inputs (r c))
    (resources (cycle))
    (operators (float fix inot idiv iadd ineq ior ileq ilgt illeq ilneg ilsub iexp isum itor itand itand rev if-true if-false if-ilm if-igt if-ieq if-ine if-ile if-igc if-ieq if-ine if-ile if-ige))
  )
  ;*** 8-cycle functional unit FUS
  ;***
  (def-mach (name fu3)
    (type functional-unit)
    (delay 2)
    (inputs (r c))
    (resources (cycle))
    (operators (sub fadd fneg fnan fmax fabs fdiv fcos fsin sqrt flt fge fne fle fsa))
  )
  ;*** 4-cycle functional unit FU4
  ;***
  (def-mach (name fu4)
    (type functional-unit)
    (delay 3)
    (inputs (r c))
    (resources (cycle))
    (operators (vbase ivload ipload ipstore ipstore fpstore fpstore fpstore fpstore fpstore))
  )
  (def-resource-class cycle 1)
  (def-resource-class constant 1)

(defun sm.immediate-constant? (constant)
  (if (inump constant)
      (and (< constant 2047)
           (> constant -2048))
      (not 0 constant))
      (consp constant)
    (equal 'address (car constant)))
  (t)
)

(defun load-constant? (constant)
  (if (sm.immediate-constant? constant)
      (not 'load immediate-constant-action)
    (else)
      t)
)
The ELI model used by Rutts CG experiments.

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

;; Register bank CIR
;;
(def-me (name (-- ci 'r) )
  (type register-bank)
  (size 64)
  (inputs (-- ci-3 'r) (-- ci-1 'r) (-- ci-8 'r) )
  (write-ports 8)
  (read-resources (-- ci-3 'r-read) )
  (write-resources (-- ci-3 'r-write) )
  (read-resources (-- ci-1 'r-read) )
  (write-resources (-- ci-1 'r-write) )
  (read-resources (-- ci-8 'r-read) )
  (write-resources (-- ci-8 'r-write) )

;; Integer adder Cl+
;;
(def-me (name (-- ci '+) )
  (type functional-unit)
  (delay 0)
  (inputs (-- ci 'r) )
  (resources (-- ci-3 'r) (-- ci-5 'r) )
`(delay 0)
(operators (div idiv cos sin sqrt))
)

(:= em-floating-multiply
 *)
  `(delay 3)
  (operators (faul faul))
)

(:= em-floating-add
 *)
  `(delay 2)
  (operators (fsub fadd fain fadd fadd fadd))
)

(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)
  )
)
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 MEs that are first on the shortest path from ME1 to ME2; () if ME1 = ME2.

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

(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) (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) )
      ) ) )
    ) )
  ) )

(defun me:me:delay ( me1 me2 )
  (? ( (me1) 0 )
     ( (me2) 0 )
     ( t
       (:: *sp.me:me:delay (me:number me1) (me:number me2) ) ) )
)

(defun me:me:first-on-path ( me1 me2 )
  (? ( (me1) )
     ( (me2) )
     ( t
       (:: *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 first-j (me-universe:number first-j) )
            (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 s) )
            )
          )
  ) )

PS:<C.S.BULLDOG.LIST-SCHEDULER.TEST>SHORTEST-PATH.LSP.18
(loop (for (top-name bottom-name) in names) (do
  (msg "!" (jc top-name -4))
  (msg 0))
(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 1)
    (left-name right-name) (sp.name:splite (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:name:delay me-i me-j))
(do (if val (then
    (if (inum val) (then
      (msg (j val -4) =)
    (else
      (msg (f val 4 2) =)))
  (else
    (msg (jc "-" -5))))
  (msg t))))
)

###========================================
### (sp.name:splite name split-<c=8?)
### Auxiliary function used by LS.DELAY-TABLE:PRINT to produce pretty
### truncated labels for the margins.
###========================================

(defun sp.name:splite (name split-<c=8?)
  (if (<= (stlength name) 4)
    ('(name =))
  (if (<= (stlength name) 8)
    (if split-<c=8?
      ('(substring name 1 4)
        (substring name 5))
    ('(name =))
  (t
    ('(substring name 1 4)
      (substring name 4)))))

PS:<C.S.BULLDOG.LIST-SCHEDULER.TEST>SHORTEST-PATH.LSP.18
ELI MACHINE-MODEL
A simple-minded, unrealistic ELI with 4 memory clusters, 4 floating
clusters, almost all parameters hardwired.

(alias "" ""rodconc)
(operators (fadd fdiv fexp cos sin sqrt tan))

(def-resource-class ,('fl '0) 1)
(def-bus ,('mi 'r) ,('fl 'r) 1)
(def-bus ,('mi+1 'r) ,('fl 'r) 1)

(eval
 'def-machine-model
   ,(loop (incr i from 0 to 8) (splice
     (cm.make-memory-cluster i 4) )
   ,(loop (incr i from 0 to 8) (splice
     (cm.make-floating-cluster i 4) )
   )
)

PS:<.C.S.BULLDOG.LIST-SCHEDULER.TEST>SIMPLE-ELI-MODEL.LSP.3
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:

```(ae operator . operands)```

`AE` 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-ae register name)` to access a register;
- `(constant-generator-ae 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 value forms.

A sample operation:

```(MO* IADD (MOR 2 X) (NOR 4 (VBASE V) ) (MOC 1) )```  

A continuation operation (a "pusher") for a multi-cycle pipeline is represented as:

```(FS+)```  

A pseudo-operation doesn't a () ME:

```(() LABEL L33)```  

```(NIL DCL %F1 FLOAT 256 ((1 266)) 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-SPEC'S LOCK-IN-REGISTERS)

`VAR-SPEC'S` is a list of program variables; prints out each variable.

---

**SIM.OPERATION-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.

---

**SIM.OPERATION-COUNT**

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 this name is reached.
"
)

(declare (special
  *sim.pcs*
  *sim.running*  
  *sim.instr*  
  *sim.oper*
  *sim.cycle*

 These globals are mainly for debugging --
 when a breakpoint occurs, you can look at
 the current cycle, instruction, and
 operation.

---

1

PS:<C.S.BULLDOG.LIST-SCHEDULER.TEST>SIMULATOR.LSP.102
The pc to start execution at (where the DEF-BLOCK is).

True if the current instruction contained a non-pseudo-op.

# operations initiated this cycle.

# COPIES executed this cycle.

```
(defun simulate (program actuals)
  (registers.initialize)
  (initialize-memory)
  (initialize-variables)
  (initialize-labels)
  (select-def-block ()
  (program:declare program)
  (if (def-block) (then
    (forms:actuals:bind (oper:part (in-variables) actuals))
  )
  (program:execute (start-pc))
  (print-program-variables (out-variables)
  )

  (select-def-block)

  (do-inline trace?
    (function (trace-var)
      (trace-var
t  

  (select-def-block)
  (allocate (length)
    (let (base
      (nax-length (vector-length memory))
      (next-free-address (+ base length))
      (if (> next-free-address max-length)
        (then
          (vector:copy 2 max-length)
          (loop (incr i from max-length to (- (vector-length memory) 1))
            (do (:= (memory i 0))))
  )

  (address:coerce (addr)
    (if (inump addr)
      (+ addr 0)
      (else
        0))

```

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:COERCED ADDR)

Forces 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).

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 zeros.

(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*
) )

;def-structure 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 ()
  (:= (hash-table:create 'eql ())) )

(defun 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:dimsions (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 (hash-table:create 'eql ()))
    (assert (== length (sim.vd:length nd))
      (== dimensions (sim.vd:dimsions nd))
      (h name) * (h length) * (h dimensions) * t
      (h nd) t)
  )

"SIMULATOR: Name previously declared with other attributes."

(defun sim.variable:initialize ( variable initial-values )
  (error (variable "SIMULATOR: Variable isn't declared." )

(defun sim.label:declare ( label pc )
  (error (list variable "SIMULATOR: Variable isn't declared." )
)

(defun sim.label:pc ( label )
  (error (list variable "SIMULATOR: Variable isn't declared." )
)

(defun sim.initialize-labels ()
  (hash-table:create () )
)

(defun sim.label:declare ( label pc )
  )
(assert t (list (asert (I (h *sim.label:pc label) label) )
:label "SIMULATOR: Label already declared."))

(defun sim.label:pc (label)
  (let* ((pc (h *sim.label:pc label) pc)
    (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
; name, an error is raised.

(defun sim.operand:declare (operand)
  (let* ((cg name operand)
    (if (and (= 2 (length operand))
      (equal :constant-generator (me:type (name:me cg)))
    (cons name (address (car name))))
      (error (list name "SIMULATOR: Invalid operand."))))

(defun sim.operand:value (operand)
  (let* ((me (me:type (name:me operand)))
    (caseq (car constant)
      (vbase address)
      (me:variable:base (cdr constant)))
    (if (consp constant)
      (error (list operand "Case error."))
      (let ((register-bank (break-point simulator))
        (if (= name (me:value-names me register))
          (me:variable:declare name 1 () name)
        (numberp name)
          (me:variable:declare name 1 () name)
        (consp name)
          (me:variable:declare name 1 (cons name))
        (error (list name "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,
; its 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)
; Records 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)
; Records 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* )

PS:<C.S.BULLDOG.LIST-SCHEDULER.TEST>SIMULATOR.LSP.102
(defun sim.initialize-pipelines ()
  (let ((cycle (+ sim.cycle delay))
        (operand (sim.next-saved-register-operand))
        (value (sim.next-saved-register-value)))
    (let* ((register (funcall *ala.saved-register-cycle* cycle))
           (name (funcall *ala.saved-register-name* register))
           (value (funcall *ala.saved-register-value* value)))
      (if (sim.trace* sim.trace-values) (then
          (nsg 0 (j operand -20) * = * value t) ) ) )
    (result (:= *ala.next-saved-register-slot* next-slot) ) )
)

(loop (initial next-slot 0)
  (incr 1 from 0 to (- *sim.next-saved-memory-slot* 1))
  (bind cycle ([] *sim.saved-memory-cycles* 1) address ([] *sim.saved-memory-addresses* 1) value ([] *sim.saved-memory-values* 1))
  (do (if (= cycle *sim.cycle*) (then
        (let ((operand (car *sim.pce*)))
          (do (sim.instr:declare-pass-1 instr) )
        (next *sim.pce* (cdr *sim.pce*)))
      (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 ) ) )
    ) )
)

(loop (initial next-slot 0)
  (incr 1 from 0 to (- *sim.next-saved-memory-slot* 1))
  (bind cycle ([] *sim.saved-memory-cycles* 1) address ([] *sim.saved-memory-addresses* 1) value ([] *sim.saved-memory-values* 1))
  (do (if (= cycle *sim.cycle*) (then
        (let ( (me-name register name) operand)
          (do (sim.instr:declare-pass-1 instr) )
        (next *sim.pce* (cdr *sim.pce*)))
      (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 ) ) )
    ) )
)

(defun sim.prograa:declare ( program )
  (:= *ala.largest-trace* 1)
  (:= *sia.8tart-pc* program)
  (loop (for oper ln instr) (do (sim.oper:declare-pass-1 oper) ) )
)

(defun sim.instr:declare-pass-1 ( instr )
  (loop (for oper in instr) (do (sim.oper:declare-pass-1 oper) ) )
)
This is the main execute loop, executing instructions until we run off the end of the program or we hit a STOP.
**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.*

```lisp
(defun sim.functional-unit-me:oper:execute ( me ((() operator . operands) ) )
  (:= *sim.significant-instruction* t)
  (if () (then () )
    (else (:= *sim.operation-count* (+ &2 &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
        (vload (vload)
          (let ( (dest Index) operands) )
            (sim.delay:register-operand:value:store delay dest (sim.address:coerce (sim.operand:value Index))))))
        (t (error (list *sim.oper* "Case error.") ) ) ) )
    (vstore (caseq operator
      (vstore (vstore)))))
```

**FUNCTIONAL-UNIT-RE:OPER:EXECUTE RE OPER**

*Executes a pseudo-operation for functional unit RE. 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.*

```lisp
(defun sim.functional-unit-re:oper:execute ( me ((() operator . operands) ) )
  (:= *sim.significant-instruction* t)
  (if () (then () )
    (else (:= *sim.operation-count* (+ &2 &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
        (vload (vload)
          (let ( (dest Index) operands) )
            (sim.delay:register-operand:value:store delay dest (sim.address:coerce (sim.operand:value Index))))))
        (t (error (list *sim.oper* "Case error.") ) ) )
    (vstore (caseq operator
      (vstore (vstore)))))
```
(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) )
(ein.operand:value source)

(error (list *sim.oper* "Case error.") ) 

(lif-coapare
  (lf (I »alB.pc«) (then
  (lot* ( (operandi operands labell label2) operanda) (label (funcall function
  (alin.operand:value operandi) (ala.operand :value operand2) labell label2) ) )
  (if label (then
  (:= »ala.pc* (sln.label:pc label))))))))

(lf-booleaa
  (lf (l »Bia.pc*) (then
  (let*( ( (operandi labell label2) operanda) (label (funcall function
  (aln.operand:value operandi) labell label2) ) )
  (lf label (then
  (:= »Bia.pc* (aln.label:pe label))))))))

(error (Hat foraal actual "Case error.") ) 

(defun ela.foraals:actuals:bind ( foraals actuals )
  (assert (llatp foraala) )
  (assert (llatp actuals) )
  (loop (for foraal ln foraals)
    (for actual ln actuals)
    (do
      (? ;*** A non-complex scalar or array? ;•••
        (assert (llatp foraal) )
        (ala.variable:initialize foraal actual) )
      ;*** A complex scalar?
      ;•••
      (assert (consp foraal) )
      (ss 3 (length foraal) )
      (for-avery (var ln foraal) (assert (llatp var) )
        (consp actual) )
      (ss 2 (length actual) )
      (numberp (car actual) )
      (numberp (cadr actual) )
      (ala.variable:initialize (cadr foraal) (car actual) )
      (ala.variable:initialize (caddr foraal) (cadr actual) ) )
      ;*** A complex array?
      ;•••
      (assert (consp foraal) )
      (for-every (var ln fornal) (assert (llatp var) )
        (consp actual) )
      (for-every (pair ln actual) (assert (consp pair) )
        (numberp (car pair) )
        (numberp (cadr pair) ) )
      (ala.variable:Initialize (cadr foraal) (loop (for (real laag) ln actual) (aave real) )
        (ala.variable:Initialize (eaddr foraal) (loop (for (raal laag) ln actual) (aave mag) )
  (t (error (Hat foraal actual "Case error.") ) ) )

(EXECUTION RECORDS
  A SIM.ER records the execution statistics of a trace (or the entire
  program).
  (def-struct sim.er
    (nuaber 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) ; # COPT operations.
    (ps-instructions 0) ; The fields keep the same counts for split- and
    (ps-operations 0) ; and join- partial schedules; these counts are
    (p-instructions 0) ; included in the totals above.
    (p-operations 0) ;
  )
  ;=======================================================================
  ;** EXECUTION RECORDS
  ;** A SIM.ER records the execution statistics of a trace (or the entire
  ;** program).
  ;**
  ;**=======================================================================
  ;** (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.fornaals:actuals:bind ( formaals actuals )
  (assert (listp formaals) )
  (assert (listp actuals) )
  (loop (for formal in formaals)
    (for actual in actuals)
    (do
      (? ;*** A non-complex scalar or array? ;**
  
PS:<C.S.BULLDOG.LIST-SCHEDULER.TEST>SIMULATOF
**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.

```lisp
(defun &initialize-execution-records ()
  (:= *sim.current.trace* 1)
  (:= *sim.last.trace* 0)
  (:= *sim.current.trace-type* 0)
  (:= *sim.trace:er* (makevector (+ 1 *sim.largest.trace*)))

(defun &record-instruction-execution ()
  (if *sim.significant.instruction* (then
      (let ((er ([*sim.trace:er* *sim.current.trace*])))
        (:= (*sim.er.instructions er) (+ 1 (*sim.er.instructions er)))
        (:= (*sim.er.operations er) (+ 1 (*sim.er.operations er)))
        (:= (*sim.er.copies er) (+ 1 (*sim.er.copies er)))
        (:= (*sim.er:entries er) (+ 1 (*sim.er:entries er)))
        (:= (*sim.er:ps.instructions er) (+ 1 (*sim.er:ps.instructions er)))
        (:= (*sim.er:ps.operations er) (+ 1 (*sim.er:ps.operations er)))
        (:= (*sim.er:pj.instructions er) (+ 1 (*sim.er:pj.instructions er)))
        (:= (*sim.er:pj.operations er) (+ 1 (*sim.er:pj.operations er)))

  (caseq *sim.current.trace-type* 
    (split (:= (*sim.er:ps.instructions er) (+ 1 (*sim.er:ps.instructions er)))
             (:= (*sim.er:ps.operations er) (+ 1 (*sim.er:ps.operations er)))
             (:= (*sim.er:pj.instructions er) (+ 1 (*sim.er:pj.instructions er)))
             (:= (*sim.er:pj.operations er) (+ 1 (*sim.er:pj.operations er)))
             ())))

(defun &print-execution-statistics ()
  (let ((sorted.ers (sort *sim.all.ers*
    (f:1 (eri er2 )
      (> (*sim.er.instructions er1) (*sim.er.instructions er2)))))
    (loop (for er in *sim.all.ers*)
      (do
        (:= (*sim.er.instructions er) (+ 1 (*sim.er.instructions er)))
        (:= (*sim.er.operations er) (+ 1 (*sim.er.operations er)))
        (:= (*sim.er.copies er) (+ 1 (*sim.er.copies er)))
        (:= (*sim.er:entries er) (+ 1 (*sim.er:entries er)))
        (:= (*sim.er:ps.instructions er) (+ 1 (*sim.er:ps.instructions er)))
        (:= (*sim.er:ps.operations er) (+ 1 (*sim.er:ps.operations er)))
        (:= (*sim.er:pj.instructions er) (+ 1 (*sim.er:pj.instructions er)))
        (:= (*sim.er:pj.operations er) (+ 1 (*sim.er:pj.operations er))))

  (let ((sorted-ers (sort *sim.all.ers*
    (f:1 (eri er2 )
      (> (*sim.er.instructions er1) (*sim.er.instructions er2)))))
    (loop (for er in *sim.all.ers*)
      (do
        (:= (*sim.er.instructions er) (+ 1 (*sim.er.instructions er)))
        (:= (*sim.er.operations er) (+ 1 (*sim.er.operations er)))
        (:= (*sim.er.copies er) (+ 1 (*sim.er.copies er)))
        (:= (*sim.er:entries er) (+ 1 (*sim.er:entries er)))
        (:= (*sim.er:ps.instructions er) (+ 1 (*sim.er:ps.instructions er)))
        (:= (*sim.er:ps.operations er) (+ 1 (*sim.er:ps.operations er)))
        (:= (*sim.er:pj.instructions er) (+ 1 (*sim.er:pj.instructions er)))
        (:= (*sim.er:pj.operations er) (+ 1 (*sim.er:pj.operations er))))

  (defun &print-heading ()
    (let ((pc (*sim.number er))
      (msg 0 t "EXECUTION PROFILE" t)
      (sim.er:print-heading)
      (msg (jc "-" 70 "-" t))
      (sim.er:print-t er t er)
      (msg (jc "-" 70 "-" 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 8)) "./", "/")))))
      (msg (jc "-" 70 "-" t))

  (defun &print (or t er pc)
    (msg (jc (sim.number er) 8 pc)
      (defun &print-heading ()
        (msg (jc "Ir" 8)
          (jc "Instrs" 7)
          (jc "Ns" 4)
          (jc "Ops" 7)
          (jc "N" 4)
          (jc "Copies" 7)
          (jc "N" 4)
          (jc "PS-I" 8)
          (jc "N" 8)
          (jc "Pf-I" 8)
          (jc "N" 5)
          (jc "Entries" 8)
          (jc "N" 5)
          (jc "D/1" 5)
          (jc "N" 5)
          (jc "C/I" 5)
          (jc "N" 5)
          (defun &print (or t er pc)
            (msg (jc (sim.number er) 8 pc)
              ...
              ...)...)
(defun X ( x y )
  (if (= y 0)
      0
      (* x 100.0))
)

(defun Xr ( x y )
  (round (X x y) )
)

(defun //f ( x y )
  (if (= y 0)
      0
      (/ x y))
)

(defun ala.var-print-program-variable ( var-apeca look-ln-registers? )
  (loop (for var-apee in var-apace) (do
    (ala.var-spac:print var-apee look-in-reglatera?) )
  )
)

(defun svnr var-spece
  '(sla.var-print-program-variable 'var-spece t) )

(defun ala.var-apee:print ( var-apee look-ln-registers? )
  (when (and look-in-registers? (not (= 0 var-apee)) )
    (ala.var:find-reglsters var-apee)
    (consp var-apee)
    (ala.var:real:laag:print var-apee)
  )
)

(defun sim.var:find-reglsters ( var )
  (loop (for me in 'sim.register-bank-mss)
    (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 ")
                (let ( (msg (t 20) "Variable " var )
                  (:= found-one? t) )
                ))
              )))
    )
  )
)

(defun sim.var:real:imag:print ( var real imag )
  (let ( (real-base (sla.variable:base real) )
      (imagic-base (lf imag (sim.var:base imag) () ) )
    (if (! (sim.var:dimensions real) )
      (do
        (msg 0 "Variable " var ")
        (sim.real:imag:offset:value real-base imag-base 0 t)
      )
      (else
        (let ( (real-en (real-var:dimensions real) )
          (imag-en (imag-var:dimensions imag) )
          (real-base (real-var:base real) )
          (imag-base (imag-var:base imag) )
        )
        (if (! (= real-en imag-en) )
          (do
            (if (simpvar:dimensions (real-var:dimensions real) )
              (do
                (when (not (= real-en imag-en) )
                  (nag "Variable " var " correspond to the dimensions of the array."
                  )
                )
              )
            )
          )
        )
      )
    )
    (let ( (dlaeneione () )
      (loop (for (lower upper)
        in (reverse (sim.var:dimensions real) )
        )
      )
    )
    (let ( (first-value '(unique-garbage)
      (final-1 (- (ala.variable:length real) 1) )
    )
      (cond
        (when (not (= first-value last-1)
          (do
            (if first-1 (then
              (msg 
              )
            )
            (if (> last-1 first-1) (then
              (sim.index:print first-1 var dimensions)
            ) (msg ")" ) )
            (sim.index:print last-1 var dimensions)
          )
        )
      )
    )
  )
)

(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?)
  )
)
)

(defun sim:var-spec:print ( var-spec look-in-registers? )
  (if (simpvar:look-in-registers?)
      (do
        (do
          (consp var-spec)
          (car var-spec)
          (cadr var-spec)
          (caddr var-spec)
          (t)
        )
        (t)
      )
  )
)

(defun sim:var:find-reglsters ( var-spec )
  (ala.var:find-reglsters var)
  (consp var-spec)
  (ala.var:real:laag:print var-spec)
  (t)
)

  (loop (for var-spec in var-specs) (do
    (sim:var:print-program-variables var-specs look-in-registers?)
  )
)
)

(defun sim:var:real:imag:print ( var real imag )
  (let ( (real-base (sim.var:base real) )
      (imagic-base (lf imag (sim.var:base imag) () ) )
    (if (! (sim.var:dimensions real) )
      (do
        (msg 0 "Variable " var ")
        (sim.real:imag:offset:value real-base imag-base 0 t)
      )
      (else
        (let ( (real-en (real-var:dimensions real) )
          (imag-en (imag-var:dimensions imag) )
          (real-base (real-var:base real) )
          (imag-base (imag-var:base imag) )
        )
        (if (! (= real-en imag-en) )
          (do
            (when (not (= real-en imag-en) )
              (nag "Variable " var " correspond to the dimensions of the array."
            )
          )
        )
      )
    )
    (let ( (dlaeneione () )
      (loop (for (lower upper)
        in (reverse (sim.var:dimensions real) )
        )
      )
    )
    (let ( (first-value '(unique-garbage)
      (final-1 (- (ala.variable:length real) 1) )
    )
      (cond
        (when (not (= first-value last-1)
          (do
            (if first-1 (then
              (msg 
            )
            )
            (if (> last-1 first-1) (then
              (sim.index:print first-1 var dimensions)
            ) (msg "") )
            (sim.index:print last-1 var dimensions)
          )
        )
      )
    )
  )
)

(defun sim:var:find-reglsters ( var )
  (loop (for me in 'sim.register-bank-mss)
    (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 ")
                (let ( (msg (t 20) "Variable " var )
                  (:= found-one? t) )
                ))
              )))
    )
  )
)

(defun sim:var:real:imag:print ( var real imag )
  (let ( (real-base (sim.var:base real) )
      (imagic-base (lf imag (sim.var:base imag) () ) )
    (if (! (sim.var:dimensions real) )
      (do
        (msg 0 "Variable " var ")
        (sim.real:imag:offset:value real-base imag-base 0 t)
      )
      (else
        (let ( (real-en (real-var:dimensions real) )
          (imag-en (imag-var:dimensions imag) )
          (real-base (real-var:base real) )
          (imag-base (imag-var:base imag) )
        )
        (if (! (= real-en imag-en) )
          (do
            (when (not (= real-en imag-en) )
              (nag "Variable " var " correspond to the dimensions of the array."
            )
          )
        )
      )
    )
    (let ( (dlaeneione () )
      (loop (for (lower upper)
        in (reverse (sim.var:dimensions real) )
        )
      )
    )
    (let ( (first-value '(unique-garbage)
      (final-1 (- (ala.variable:length real) 1) )
    )
      (cond
        (when (not (= first-value last-1)
          (do
            (if first-1 (then
              (msg 
            )
            )
            (if (> last-1 first-1) (then
              (sim.index:print first-1 var dimensions)
            ) (msg "") )
            (sim.index:print last-1 var dimensions)
          )
        )
      )
    )
  )
)

(defun sim:var:find-reglsters ( var )
  (loop (for me in 'sim.register-bank-mss)
    (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 ")
                (let ( (msg (t 20) "Variable " var )
                  (:= found-one? t) )
                ))
              )))
    )
  )
)

(defun sim:var:real:imag:print ( var real imag )
  (let ( (real-base (sim.var:base real) )
      (imagic-base (lf imag (sim.var:base imag) () ) )
    (if (! (sim.var:dimensions real) )
      (do
        (msg 0 "Variable " var ")
        (sim.real:imag:offset:value real-base imag-base 0 t)
      )
      (else
        (let ( (real-en (real-var:dimensions real) )
          (imag-en (imag-var:dimensions imag) )
          (real-base (real-var:base real) )
          (imag-base (imag-var:base imag) )
        )
        (if (! (= real-en imag-en) )
          (do
            (when (not (= real-en imag-en) )
              (nag "Variable " var " correspond to the dimensions of the array."
            )
          )
        )
      )
    )
    (let ( (dlaeneione () )
      (loop (for (lower upper)
        in (reverse (sim.var:dimensions real) )
        )
      )
    )
    (let ( (first-value '(unique-garbage)
      (final-1 (- (ala.variable:length real) 1) )
    )
      (cond
        (when (not (= first-value last-1)
          (do
            (if first-1 (then
              (msg 
            )
            )
            (if (> last-1 first-1) (then
              (sim.index:print first-1 var dimensions)
            ) (msg "") )
            (sim.index:print last-1 var dimensions)
          )
        )
      )
    )
  )
)
(defun sim.index:print (i var dimensions)
  (loop (for (lower upper size) in dimensions)
    (initial remainder 1)
    (do
      (setq * = (+ 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 (values me-name) )
    ',( [] me:values me register)
      ,[] me:value-names me register) )

PS:<C.S.BULLDOG.LIST-SCHEDULER.TEST>SIMULATOR.LSP.102
TEST BED

This module implements a test bed that allows the use of "trace snapshots" 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]

Invoke 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)

---

(defun tls

  (symbol-table live-before source-record-list live-after
  predecessors-results)

  &optional (print? t)

  (:= *tb.symbol-table* symbol-table)

  (:= *tb.index:datum* (makevector (length source-record-list) ))

  (:= *tb.index:result* (makevector (length source-record-list) ))

  (vector:initialize *tb.index:results predecessors-results)

  (:= *tb.current-index*

    (if (= 'def (operator (car (car source-record-list) ) )

     1

     0 )

     := *tb.current-index*

    )

  )

  (:* * (generate-code live-before source-record-list live-after)

    (if print? (then

      (sch.print-schedule) )

    )

  )

  )

---

(START-TRACE)

---

(PREDECESSORS SOURCE-OPER TRACE-DIRECTION DATUM)

---

(defun start-trace ()

  ()

  )

---

(defun predecessors ( source-ope oper trace-direction datum )

  (let* ( result

        (loop (for (index . reasons) in ([] *tb.index:results

          (*tb.index:datum* index)

          ,.reasons) ) )

        )

    )

  )

---

1 PS:<C.S.BULLDOG.LIST-SCHEDULER.TEST>TEST-BED.LSP.17
(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))
        ....
        (defun test-bed.undo ()
          (loop (for (name def) in *tb.saved-names&defs*)
                (do ((:= (fundef name) def)) ) )
          () )
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ....
        ...
**VALUE NODES**

A VN (Value Node) represents one value (and possibly an operation) in the expression DAG for a trace.

```lisp
(defun vn
  (number  ; A unique number; guaranteed to produce source ordering of nodes in the trace.
  type    ; One of {'DEF, 'DEFI, 'OPERATION, 'USE, 'USEI, 'COPY, 'PSEUDO-OP}
  constant? ; True if a constant DEF.
  datatype ; Datatype of this value, one of {'INTEGER, 'FLOAT}
  data    ; The opaque bookkeeper data 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 :CONSTRAIN-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, analogous to the :DEPTH.
  me ; The functional unit or constant generator where this VN will be computed.
)
```

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:

```lisp
  (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 one 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.

---

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 
  (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:operator (VN)
  (assert (VN:IS VN))
  (oper:operator (VN:OPER VN)))

(defun VN:splice-VN (VN NEW-VN READING-VNS OFF-LIVE-READING-VNS)
  (assert (subset? reading-VNS (VN:OPERAND-VNS VN)))
  (assert (subset? off-live-reading-VNS (VN:OPERAND-VNS VN)))
  (assert (! reading-VNS off-live-reading-VNS))
  (:= (VN:OPERAND-VNS VN) (set-diffq & & & reading-VNS)
      (push (VN:OPERAND-VNS VN) new-VN)
      (:s (VN:OPERAND-VNS VN) , (VN)))
  (:= (VN:OPERAND-VNS VN) (VN:OPERAND-VNS VN))
  (loop (for reading-VN in reading-VNS) (do
     (:s (VN:OPERAND-VNS VN) , (TOP-LEVEL-SUBSTQ new-VN)))
     (:s (VN:OPERAND-VNS VN) , (TOP-LEVEL-SUBSTQ new-VN)))
  (:= (VN:OPERAND-VNS VN) (set-diffq & & & off-live-reading-VNS)
      (push (VN:OPERAND-VNS VN) new-VN)
      (:s (VN:OPERAND-VNS VN) , (VN)))
  (:= (VN:OPERAND-VNS VN) (VN:OPERAND-VNS VN))
  (loop (for off-live-reading-VN in off-live-reading-VNS) (do
    (:s (VN:OPERAND-VNS OFF-LIVE-READING-VNS) , (VN:OPERAND-VNS OFF-LIVE-READING-VNS))
    (:s (VN:OPERAND-VNS OFF-LIVE-READING-VNS) , (VN:OPERAND-VNS OFF-LIVE-READING-VNS)))
  )
)
(defun vn:delete (vn)
 (assert (v2 (vn:reading-vns vn))
 (v1 (vn:off-live-reading-vns vn))
 (v1 (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 (v2 constraining-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)
 (:dreverse new-constraining-delays constrained-vn)
 ))
 )

;*** Now removes references to this VN from constraining VNs.
;
;***
(loop (for constraining-vn in (vn:constraining-vns vn)) (do
 (:d=(vn:constraining-vns vn)
 top-level-removq vn)
 (loop (for constraining-vn in (vn:constraining-vns vn))
 (for constraining-delay in (vn:constraining-delays vn))
 (do
 (vag constraining-vns vn)
 (vag constraining-delays vn)
 ))
)

;*** Remove all references to VN from its operands.
;
;***
(loop (for operand-vn in (vn:operand-vns vn)) (do
 (result (vn:reading-vns operand-vn)
 top-level-removq vn)
 )
)

(vn-universe:delete vn)
();

(defun vn:print (vn)
 (assert (vn:ls vn)
)
 (casex (vn:type vn)
 (def (msg "def " (vn:name vn) " ")
 (def1 (msg "def1 " (vn:name vn) " ")
 (use (msg "use " (vn:name vn) " ")
 (use1 (msg "use1 " (vn:name vn) " ")
 (operation (msg (vn:operation vn) " ")
 (pseudo-op (msg (vn:operation vn) " ")
 (copy (msg "copy " (vn:operation 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) " ")
 ))
 ))
)
)
)

(if (vn:constraining-vns vn) (then
 (msg " constrained-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:constraining-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-nes vn) (then
 (msg " likely-nes:"
 (h (le:name-list (vn:likely-nes vn))) t)
 ))

(if (vn:locations vn) (then
 (msg " locations:"
 (h (loop (for me register in (vn:locations vn)) (save
 (aeo:save register) )
 t)
 ))

(if (vn:register-bank-me vn) (then
 (msg " register:"
 (me:name (vn:register-bank-me vn) t"
 (vn:register vn)
 ))
)
)

PS:<C.S.BULLDOG.LIST-SCHEDULER.TEST>VN.LSP.24
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 "IM.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 48)
  (= :dag:node-width 100)
  (= :dag:line-width 25)
  (= :dag:box-width 80)
  (= :dag:min-level-separation 30)
  (= :dag:ideal-text-lines-per-node 4)
  (= :dag:fold-text-lines* 0)
  (= :dag:critical-threshold* 0)
  (= :dag:remove-non-critical-nodes-edges?)
  (= :dag:shade-critical-nodes?*)
  (= :dag:text-x-margin* 8)
  (= :dag:make-slides* )))

(defun write-vns (optional (filename "im.dag") dag-commands)
(let ((filename (file-list:filename
(file-list:merge filename '() () "DAG") )))
  (tota ( (file filename "out newversion") ) (without file
  (msg h 
    ,(*wv.default-dag-commands* ,dag-commands)
  00000 00000) t)
  (msg h (wv:dag:graph) 00000 00000) t)
  (filename file) ) ))

(defvar *wv.dag:graph* ()
  (let (*wv.all-nodes* ()
    (for each-vn vn)
    (when (wv:visible? vn)
      (do
        (push *wv.all-nodes*
          (case 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)
          )
        )))
      ))
  )))

(defvar *wv.default-usage-vn:* (wv:default-usage-vn:create-node vn)
  (copy
    (wv:copy-vn:create-node vn) ))
  (result
    (reverse *wv.all-nodes*) ))

(defvar *wv.def-use-vn:create-node ( vn )
  (dag:input-node:nev
    name (vn:name vn)
    label "((string-sug (vn:name vn) " (vn:name vn) )
      (wv:fn:location vn))
    style 'shading2
    child-edge-styles
      (wv.vn:child-edge-styles vn))

(defvar *wv.defi-usei-vn:create-node ( vn )
  (dag:input-node:nev
    name (vn:name vn)
    label "((string-sug (vn:name vn) " (vn:name vn) )
      (wv.vn:location vn))
    style 'shading2
    child-edge-styles
      (wv.vn:child-edge-styles vn))

(defvar *wv.operation-vn:create-node ( vn )
  (dag:input-node:nev
    name (vn:name vn)
    label "((string-sug (vn:name vn) " (vn:name vn) )
      (wv.vn:location vn))
    style 'shading2
    child-edge-styles
      (wv.vn:child-edge-styles vn))

(defvar *wv.copy-vn:create-node ( vn )
  (dag:input-node:nev
    name (vn:name vn)
    label "((string-sug (vn:name vn) " (vn:name vn) )
      (wv.vn:location vn))
    style 'shading2
    child-edge-styles
      (wv.vn:child-edge-styles vn))

(defvar *wv.vn:me-name ( vn )
  (( (vn:register-me vn)
      (wv.vn:me-name vn)
    )
  )))
(defun w.vn:location (vn)
  (if (vn:register-bank-me vn)
      (string-ag (me:name (vn:register-bank-me vn)) =
                 (vn:register vn))
    (vn:me vn)
    (me:name (vn:me vn))
    (t ""))
)

(defun w.vn:child-edge-styles (vn)
  (aconc
    (loop (for reading-vn in (noduplesq (vn:reading-vns vn))
           (when (w.vn:visible? reading-vn)
                 (w.vn:child-vn:style:create-edge vn reading-vn 'thick)))
    (loop (for constrained-vn in (noduplesq (vn:constraining-vns vn)))
           (when (& (vn:constraining-vn:delay vn constrained-vn) 0)
                 (w.vn:visible? constrained-vn)
                 (w.vn:child-vn:style:create-edge vn constrained-vn 'shaded)))
  ))

(defun w.vn:visible? (vn)
  (caseq (vn:type vn)
    (pseudo-op (def (for-some (reading-vn in (vn:reading-vns vn))
                       (not (!= 'use (vn:type reading-vn)))
                       (w.vn:visible? reading-vn))
                (t (error (list vn "Case error."))) )
    (def (for-some (reading-vn in (vn:reading-vns vn))
                (w.vn:visible? reading-vn)))
    (t (error (list vn "Case error."))) )
  ))

(defun w.vn:create-chain
  (length type label box-style edge-style child-edge-styles)
  (loop (initial prev-child-edge-styles child-edge-styles)
        (decr 1 from length to 1)
        (bind node (dag:input-node:new
                     name
                     type
                     label
                     style
                     box-style
                     child-edge-styles
                     prev-child-edge-styles)
        (do
          (push w.vn:all-nodes node
            #:prev-child-edge-styles
            #:edge-style)
          (result prev-child-edge-styles)
        ))
)

(defun w.vn:next-cycle (vn)
  (+ 1 (+ (w.vn:cycle vn)
            (caseq (vn:type vn)
                   (operation (se:delay (vn:me vn))
                               (t 0)))
       )
)

(defun w.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))
    (use (vn:cycle vn)
          (if (= 'use1 (vn:type (car (vn:operand-vns vn))))
              (+ 1 (vn:cycle vn))
              (vn:cycle vn))
          )
    )
    (error (list vn "Case error."))) )

(defun w.vn:visible? (vn)
  (caseq (vn:type vn)
    (pseudo-op ()
      (def (for-some (reading-vn in (vn:reading-vns vn))
               (as (im 'use (vn:type reading-vn))
                    (w.vn:visible? reading-vn))
                   (def1 (for-some (reading-vn in (vn:reading-vns vn))
                             (as 'use (vn:type reading-vn))
                             (t (error (list vn "Case error.")))
                             )
                   ))
    )
    (t (error (list vn "Case error."))) )
  )

PS:<C.S.BULLDOG.LIST-SCHEDULER.TEST>WRITE-VNS.LSP.23