```
(in-package "EMU")

(setf *instruction-list*
      '(
        (%+ 2 msys::%+ )
        (%- 2 msys::%- )
        (%* 2 msys::%* )

        (%div 2 msys::%div )
        (%rem 2 msys::%rem )

        (%<= 2 msys::%<= )
        (%>= 2 msys::%>= )
        (%< 2 msys::%< )
        (%> 2 msys::%> )
        (%= 2 msys::%= )
        (%/= 2 msys::%/= )

        (JNIL 0 jnil-fun )
        (JMP 0 jmp-fun )

        (LOD 0 lod-fun )
        (LODNIL 0 lodnil-fun)

        (STO 0 sto-fun )

        (LITIDX 0 litidx-fun )

        (INCSP 0 incsp-fun )
        (DECSP 0 decsp-fun )

        (HALT 0 halt-fun)

        (CALLCLOS 0 callclos-fun)
        (CALLCLOSV 0 callclosv-fun)
        (RET 0 ret-fun)

        (LODCS 0 lodcs-fun)
        (PUSHCS 0 pushcs-fun)
        (INCCSP 0 inccsp-fun)

        (EXCEPT 0 except-fun)


        (%read 0 read-fun )
        (%print 1 print-fun)

        (%make-closure 2 msys::make-closure)
        (%make-vector 1 msys::make-vector)

        (%set-symbol-function 2 msys::set-symbol-function)
        (%set-symbol-value 2 msys::set-symbol-value)

        (%symbol-function 1 msys::symbol-function)

        (%symbol-value 1 msys::symbol-value)


        (%svref 2 msys::svref)
        (%set-svref 3 msys::set-svref)

        (%closure-ref 2 msys::closure-ref)

        (eq 2 msys::eq)
```



```
        (car 1 msys::car)
        (cdr 1 msys::cdr)
        (cons 2 msys::cons)))



(setf *num-instr* (length *instruction-list*))

(setf *dispatch* (make-array *num-instr*))


; die opcode-liste

(setf *opcode-list* nil)
(setf *instr-list-len* (length *instruction-list*))

(setf *opcode-vector* (make-array *instr-list-len*))


(defun make-opcode-vector ()
  (do ((i 0 (+ i 1))
       (p *instruction-list* (cdr p)))
      ((>= i *instr-list-len*))
    (setf (aref *opcode-vector* i) (car p))))

(make-opcode-vector)

(defun make-instr-assoc ()
  (let ((res nil))
    (do ((i 0 (+ i 1)))
        ((>= i *instr-list-len*) res)
      (setf res (cons (cons (car (aref *opcode-vector* i)) i) res)))))

(setf *instr-assoc* (make-instr-assoc))

(defun get-opcode (instr)
  (let ((subs (assoc instr *instr-assoc*)))
    (if (null subs)
        (let ()
          (format t "instruction / ~A / not found~%" instr)
          (error "instruction not found"))
      (cdr subs))))



; der virtuelle prozessor

; initialisierung

(defun emu-init (tp-init)
; program counter
(setf *pc* 0)

; data stack pointer
(setf *dsp* 65536)

; control stack pointer
(setf *csp* 32768)

(setf *halt-flagged* nil)

; template pointer
(msys::set-word 252 0 tp-init)
(setf *tp* (msys::get-word 252 0))
```



```
(setf *pc* (msys::get-word *tp* (msys::cl-to 4))))



(setf *watch-instr* nil)

(defun debug-emu (on)
  (if on
      (let ()
        (setf *watch-instr* 1))
    (let ()
      (setf *watch-instr* nil))))


(defparameter *instr-mask* (ash (- (expt 2 16) 1) 16))
(defparameter *ext-mask* 65535)


(defun instr-opc (instr)
  (ash (logand instr *instr-mask*) -16))

(defun instr-ext (instr)
  (logand instr *ext-mask*))

(defun logicalp (x)
  (or (eq x t) (eq x nil)))

(defun run-processor (max-steps)
  (let ((instr nil)
        (opc nil)
        (nargs nil)
        (args nil)
        (old-pc nil)
        (old-dsp nil)
        (old-csp nil)
        (opc-ext nil)
        (step-cnt nil))
    (do ((step-cnt 0 (+ 1 step-cnt)))
        ((or *halt-flagged* (>= step-cnt max-steps)) (format t "~%~% EMU-Steps = ~
A~%~%" step-cnt))
      (setf instr (msys::get-word *pc* 0))
      (setf opc (instr-opc instr))
      (setf opc-entry (aref *opcode-vector* opc))
      (setf opc-ext (instr-ext instr))
      (if *watch-instr*
          (format t "emu:: instr = ~x: ~A ~A~%" *pc* (car opc-entry) opc-ext))
      (setf nargs (cadr opc-entry))
      (setf args (let ((res nil))
                   (do ((i 0 (+ i 1)))
                       ((>= i nargs) res)
                     (setf res (cons (msys::get-word (+ *dsp* (* i 4)) 0) res)))
                   res))

      (setf old-pc *pc*)
      (setf old-dsp *dsp*)
      (setf old-csp *csp*)
      (setf *jmp-flag* 0)
      (if (> nargs 0)
          (let ()
            (setf res (apply (caddr opc-entry) args))
            (when (logicalp res)
;               (format t "umformung logisch:")
              (setf res (msys::log-to res))))
        (setf res (apply (caddr opc-entry) (list instr))))
```



```
      (if (> nargs 0)
          (let ()
            (setf *dsp* (+ *dsp* (* 4 (- nargs 1))))
            (msys::set-word *dsp* 0 res)))
      (if (zerop *jmp-flag*)
          (setf *pc* (+ *pc* 4))))))



(defun push-stack (val)
  (let ()
    (setf *dsp* (- *dsp* 4))
    (msys::set-word *dsp* 0 val)))

(defun pop-stack ()
  (let ()
    (setf *dsp* (+ *dsp* 4))
    (msys::get-word (- *dsp* 4) 0)))

(defun push-address-stack (val)
  (let ()
    (setf *csp* (- *csp* 4))
    (msys::set-word *csp* 0 val)))

(defun pop-address-stack ()
  (let ()
    (setf *csp* (+ *csp* 4))
    (msys::get-word (- *csp* 4) 0)))

(defun litidx-fun (instr)
  (let ((val (msys::templ-ref-d *tp* (msys::cl-to (instr-ext instr)))))
    (push-stack val)))

(defun decsp-fun (instr)
  (let ((d (instr-ext instr)))
    (setf *dsp* (- *dsp* (* 4 d)))))

(defun incsp-fun (instr)
  (let ((d (instr-ext instr)))
    (setf *dsp* (+ *dsp* (* 4 d)))))

(defun inccsp-fun (instr)
  (let ((d (instr-ext instr)))
    (setf *csp* (+ *csp* (* 4 d)))))

(defun lod-fun (instr)
  (let ((d (instr-ext instr)))
    (let ((val (msys::get-word (+ *dsp* (* 4 d)) 0)))
      (push-stack val))))

(defun pushcs-fun (instr)
  (let ((d (instr-ext instr)))
    (let ((val (msys::get-word (+ *dsp* (* 4 d)) 0)))
      (push-address-stack val))))

(defun lodcs-fun (instr)
  (let ((d (instr-ext instr)))
    (let ((val (msys::get-word (+ *csp* (* 4 d)) 0)))
      (push-stack val))))

(defun lodnil-fun (instr)
  (push-stack msys::nil))


(defun sto-fun (instr)
```

```
  (let ((d (instr-ext instr)))
    (let ((val (msys::get-word *dsp* 0)))
      (setf *dsp* (+ *dsp* 4))
      (msys::set-word (+ *dsp* (* 4 (- d 1))) 0 val))))


(defun two-comp (x)
  (if (>= x 32768)
      (- x 65536)
    x))


(defun jnil-fun (instr)
  (let ((d (instr-ext instr)))
    (let ((val (pop-stack)))
      (if (eq val msys::nil)
          (let ()
            (setf *pc* (+ *pc* (* 4 (two-comp d))))
            (setf *jmp-flag* 1))))))

(defun jmp-fun (instr)
  (let ((d (instr-ext instr)))
    (setf *pc* (+ *pc* (* 4 (two-comp d))))
    (setf *jmp-flag* 1)))

(defun halt-fun (instr)
  (setf *halt-flagged* t))

(defun read-fun (instr)
  )

(defun print-fun (val)
  (format t "EMU: ~A~%" (msys::to-cl val))
  val)

(defun except-fun (instr)
  (let ()
    (format t "*** EMU: exception occurred~%")
    (setf *halt-flagged* true)))


(defun callclos-fun (instr)
  (let ((d (instr-ext instr)))
    (let ((clos (msys::get-word (+ *dsp* (* 4 (- d 1))) 0)))
      (push-stack (msys::cl-to 4 d))
;     (format t "type clos= ~A~%" (msys::show-types clos))
      (let ((templ (msys::closure-ref clos 0)))
        (push-address-stack *tp*)
        (setf *tp* templ)
        (push-address-stack (+ *pc* 4))
        (setf *pc* (msys::get-word *tp* (msys::cl-to 4)))
        (setf *jmp-flag* 1)))))

(defun callclosv-fun (instr)
  (let ((d (msys::to-cl (msys::get-word *dsp* 0))))
    (let ((clos (msys::get-word (+ *dsp* (* 4 (+ d 1))) 0)))
;     (format t "type clos= ~A~%" (msys::show-types clos))
      (let ((templ (msys::closure-ref clos 0)))
        (push-address-stack *tp*)
        (setf *tp* templ)
        (push-address-stack (+ *pc* 4))
        (setf *pc* (msys::get-word *tp* (msys::cl-to 4)))
        (setf *jmp-flag* 1)))))
```



```
(defun ret-fun (instr)
  (let ((new-pc (pop-address-stack))
        (new-tp (pop-address-stack)))
    (setf *tp* new-tp)
    (setf *pc* new-pc)
    (setf *jmp-flag* 1)))
```