

(TECHNICAL NOTES

A VLISP Interpreter  
on the VCMC1 Machine

May 1977

Jerome CHAILLOUX

Universite de Paris VIII - Vincennes  
Route de la Tourelle  
75571 Paris Cedex 12 (France)

VCMC1 is a virtual machine designed to observe "in vitro" the behaviour of VLISP interpreters. VCMC1 is actually entirely simulated in VLISP 10. We present a short description of the VCMC1 machine followed by the complete listing of the code of a VLISP interpreter. This interpreter incorporates the special feature for tail-recursion function calls.

Basically VCMC1 is a 16 bits machine. An instruction uses one, two, three or four words, and has one, two or three operands. Each operand is coded on a 4 bits field.

There are two formats for the instructions :

3 operands instructions :

[op. code , 1st operand , 2nd operand , 3th operand]  
bits 15 .. 12 11 .. 8 7 .. 4 3 .. 0

2 operands instructions :

[op. code , 1st operand , 2nd operand]  
bits 15 8 7 .. 4 3 .. 0

specification of the operands

AX register 0. Holds sometimes the result of complex instructions (e.g. GET). It is used as index register.

- A1 register 1.
- A2 register 2.
- A3 register 3.
- A4 register 4.
- A5 register 5.
- A6 register 6.

A VLISP Interpreter on the VCMCI Machine

LINK     register 7.

PC        register 8. Is the program counter.

ST        register 9. Is the stack pointer.

TST       the top of the stack.

+TST      the top of the stack after incrementation of the stack pointer.

TST-      the top of the stack. The stack pointer is decremented after the computation of the effective address.

(value) a 16 bits value enclosed in parenthesis. This value is stored in the word just following the instruction.

(@ . address) the value contained in the location specified. Used to denote indirection. The address is stored in the word following the instruction.

NIL       the atom NIL itself.

These operands allow several kinds of addressing :

- direct on registers AX A1 A2 A3 A4 A5 A6 LINK PC ST
- indirect on SP
- auto-increment on SP and PC
- auto-decrement on SP
- auto-increment indirect on PC.

Terminology and notation

r1        effective address of the 1st operand

r2        effective address of the 2nd operand

r3        effective address of the 3th operand

r' -> r'' move the content of the effective address r' into the word of effective address r''

r' <-> r'' exchange the contents of the effective addresses r' and r''

(CAR r)   the CAR of the effective address r

(CDR r)   the CDR of the effective address r

act1 & act2 denotes the overlap of the two actions

The instruction set

Instructions described are only those used by the interpreter listed below.

The effective addresses of the two or three operands are computed before the execution of the instructions, except in the case of conditional jumps.

transfers of data

(MOVE r1 r2)    r2 -> r1

## A VLISP Interpreter on the VCMC1 Machine

```
(EXCH r1 r2)    r2 <-> r1
(MOVD r1 r2 r3) r1 -> r2 & r2 -> r3
(CAR r1 r2)     (CAR r2) -> r1
(CDR r1 r2)     (CDR r2) -> r1
(RPLACA r1 r2)  r2 -> (CAR r1)
(RPLACD r1 r2)  r2 -> (CDR r1)

(MPUSH r1 r2)   r1 -> +TST ; if r2 # NIL, r2 -> +TST
(MPOP r1 r2)    TST- -> r1 ; if r2 # NIL, TST- -> r2
```

### Transfers and branches

```
(MOVR r1 r2)    r2 -> r1 & TST- -> PC
(MOVJ r1 r2 r2) r2 -> r1 & r3 -> PC
(MOVC r1 r2 r3) r2 -> r1 & PC -> +TST & r3 -> PC
(CARR r1 r2)    (CAR r2) -> r1 & TST- -> PC
(CDRR r1 r2)    (CDR r2) -> r1 & TST- -> PC
(RPLACAR r1 r2) r2 -> (CAR r1) & TST- -> PC
(RPLACDR r1 r2) r2 -> (CDR r1) & TST- -> PC
```

### Unconditional branches

```
(JUMP r1)       r1 -> PC
(JUMPX r1 r2)   r1 + r2 -> PC
(CALL r1)       PC -> +TST & r1 -> PC
(RETURN)        TST- -> PC
```

### Conditional branches

```
(JEQ r1 r2 r3)  if r1 = r2 then r3 -> PC
(JNEQ r1 r2 r3) if r1 # r2 then r3 -> PC
(JTNIL r1 r2)   if r1 = NIL then r2 -> PC
(JTNIL r1 r2)   if r1 # NIL then r2 -> PC

(JTLIST r1 r2)  if r1 is a pointer on a list then r2 -> PC
(JFLIST r1 r2)  if r1 is not a pointer on a list
then r2 -> PC

(JTNUMB r1 r2)  if r1 is a pointer on a number then r2 -> PC
(JFNUMB r1 r2)  if r1 is not a pointer on a number
then r2 -> PC
```

### Other instructions

```
(UNCONS r1 r2 r3) (CAR r1) -> r2 & (CDR r1 -> r3)
(CONS r1 r2 r3)   (r2 , r3) -> r1
(GET r1 r2)       (GET r1 r2) -> AX
```

### Syntax of the assembler

The VLISP simulator handles lists of VCMC1 instructions, in which atomic elements are labels. It is possible to abbreviate instructions which look like

```
(opcd op1 op2 (label))
```

```
into (opcd op1 op2 . label)
```

In order to decrease the size of such a list.

(PROGN  
(SETQ -INTERPRETER '(

-----  
; V C M C 1  
; VLISP interpreter  
-----

; bindings of arguments for standard functions ;  
; ;  
; 1SUBR : A1 <- value of the 1st argument ;  
; 2SUBR : A2 <- value of the 1st argument ! ;  
; A1 <- value of the 2nd argument ! ;  
; NSUBR : A1 <- list of values of all the arguments ;  
; FSUBR : A1 <- list of all the arguments non-evaluated ;

; TOP-LEVEL function ;

TOPLEVEL

(MOVQ A1 ('TOPLEVEL) . PRINTA1) ; (WHILE T ;  
(CALL . READ) ; (PRINT 'TOPLEVEL) ;  
(CALL . EVAL) ; (PRINT ;  
(MOVJ +TST (TOPLEVEL) . PRINTA1); (EVAL (READ))) ;

; PRINTA1 : because PRINT is a system NSUBR ;

PRINTA1 (CONS A1 A1 NIL)  
(JUMP . PRINT)

-----  
; functions of the interpreter ;  
-----

; GETFN : recognizes the type of the function stored in A2 ;  
; call : (MOVJ A6 PC . GETFN) i.e. return address in A6 ;  
; result in AX ;  
; AX <- 1 if 1SUBR the address of the function is stacked ;  
; AX <- 2 if 2SUBR ; ;  
; AX <- 3 if NSUBR ; ;  
; AX <- 4 if FSUBR ; ;  
; AX <- 5 if LAMBDA ((lvar) ... body ...) is stacked ;  
; AX <- 6 if FLAMBDA ; ;  
; AX <- 7 if GAMMA ; ;  
; don't destroy A1 ! ;

GETFN (JTLIST A2 . GETFN5)

(GET A2 ('EXPR)) ; the function is an atom ;  
; is it an EXPR ? ;  
(JTNIL AX . GETFN1) ; no ;  
(MOVJ A2 AX . GETFN) ; yes : retry with the new expression ;  
GETFN1 (GET A2 ('TYPFN)) ; is it a standard function ? ;  
(JTNIL AX . GETFN3) ; no ;  
(MOVJ +TST (\*VAL\* [A2]) A6) ; yes : stack the address and return ;  
GETFN3 (CAR A2 A2) ; indirection on the value of the atom ;  
(JUMP . GETFN)

GETFN5

(UNCONS A2 AX +TST) ; the function is a list ;  
(JNEQ AX ('LAMBDA) . GETFN6) ; stack ((lvar) ... body ...) ;  
; is it a LAMBDA ? ;  
(MOVJ AX ('5) A6) ; yes : value = 5 and return ;  
GETFN6 (JNEQ AX ('FLAMBDA) . GETFN7) ; is it a FLAMBDA ? ;  
(MOVJ AX ('6) A6) ; yes : value = 6 and return ;  
GETFN7 (JNEQ AX ('GAMMA) . GETFN8) ; is it a GAMMA ? ;  
(MOVJ AX ('7) A6) ; yes : value = 7 and return ;

GETFN8

(MPUSH A1) ; in others cases ;  
(MOVQ A1 A2 . EVAL) ; re-evaluate the function ;  
(MOVD A2 A1 TST-)  
(MOVJ A6 TST- . GETFN)

; EVAL : 1SUBR A1 <- the forme to be evaluate ;  
; APPLY : 2SUBR A1 <- the list of values ready ;  
; A2 <- function to apply ;

EVALCAR (CAR A1 A1)

; (EVAL (CAR A1)) ;

EVAL (JTLIST A1 . EVAL1)

; in case of a list ;

QUOTE (JTNUMB A1 TST-)

; numbers are not evaluated ;

QUOTE (CARR A1 A1)

; the value of an atom is  
its C-value (i.e. its CAR) ;

EVAL1 (UNCONS A1 A2 A1)

; A1 <- the function ;

(JEQ A1 ('QUOTE) . QUOTE)

; special case for the QUOTE function ;

(MOVJ A6 PC . GETFN)

; find the type of the function ;

(JUMPX AX ((EVALCAR)(EVAL2)(EVLIS) TST- (EVAL3)(APPLY)(EVAL4)))

; 1SUBR 2SUBR NSUBR FSUBR LAMBDA FLAMBDA GAMMA ;

```

EVAL2  ; for the 2SUBRs ;
        (UNCONS A1 A1 +TST)
        (CALL . EVAL) ; evaluate the 1st argument ;
        (EXCH A1 TST)
        (CALL . EVALCAR) ; evaluate the 2nd argument ;
        (MOVR A2 TST-)

EVAL3  ; evaluation of LAMBDA-expressions ;
        (MOVJ +TST (APPLYL) . EVLIS)

EVAL4  ; evaluation of GAMMA-expressions ;
        (MOVJ +TST (APPLYG) . EVLIS)

APPLYC (CONS A1 A1 NIL) ; used by mapping functions ;
APPLY  (MOVJ A6 PC . GETFN) ; set the type of the function ;
        (JUMPX AX ((CAR)(APPLY2) TST- TST- (APPLYL)(APPLYG)(APPLYG)))
        ; 1SUBR 2SUBR NSUBR ESUBR LAMBDA FLAMBDA GAMMA ;
APPLY2 (UNCONS A1 A2 A1)
        (CARR A1 A1)
APPLYG (CAR A1 A1)
        (JUMP . APPLYL)
APPLYF (CONS A1 A1 NIL)
        ; APPLYL must follow ... ;

; general for LAMBDA/FLAMBDA/GAMMA ;
; suppose : A1 ← list of values ready ;
;          TST ← ((lvar) ... body ...) ;

APPLYL (UNCONS TST- A2 A3) ; A2 ← lvar, A3 ← body. ;
        ; test of tail-recursion ;
        (JNEQ (@ . TST) (*TR*) . APPLYN) ; it is not in terminal position ;
        (JNEQ (@ . LINK) A3 . APPLYN) ; it is not a recursive function ;
        ; special binding for tail-recursive function ;

REBIND (JFLIST A2 . REBIND2) ; lvar is atomic ;
REBIND1 (UNCONS A2 A5 A2) ; A5 ← new variable ;
        (UNCONS A1 (@ . A5) A1) ; force the new value ;
        (JTLIST A2 . REBIND1) ; variables left ? ;
REBIND2 (JTNIL A2 . PROGNA3) ; real end of lvar ;
        (MOVJ (@ . A2) A1 . PROGNA3) ; in case of LEXPR ;
        ; normal binding with preservation of the old values ;

APPLYN (MPUSH LINK ('MARKER)) ; special mark in stack ;
BIND1 (JFLIST A2 . BIND2)
        (UNCONS A1 A4 A1) ; A4 ← next value ;
        (UNCONS A2 A5 A2) ; A5 ← next variable ;
        (MOVD +TST (@ . A5) A4)
        (MOVJ +TST A5 . BIND1)
BIND2 (JTNIL A2 . BIND3) ; real end of lvar ;
        (MOVD +TST (@ . A2) A1)
        (MPUSH A2)
BIND3 (MPUSH A3)
        ; execution of the body of the function ;
        (MOVC LINK ST . PROGNA3)
*TR* (MOVJ A6 PC . UNBIND)
        (RETURN)
        ; unbind the previous bindings ;

UNBIND (MOVJ A5 TST- . UNBIND2)
UNBIND1 (RPLAC6 A5 LINK)
UNBIND2 (MPOP A5 LINK)
        (JNEQ A5 ('MARKER) . UNBIND1)
        (JUMP A6)

```

```

;-----;
; control functions ;
;-----;

; PROGN : FSUBR, EPROGN : 1SUBR ;
; allows to handle the tail-recursive functions ;

PROGNA3 (MOVE A1 A3) ; internal (PROGN A3) ;
EPROGN
PROGN (UNCONS A1 A1 A2) ; next element ;
(JFLIST A2 . EVAL) ; there is one element ;
PROGN1 (MOVC +TST A2 . EVAL)
(UNCONS TST- A1 A2) ; next element ;
(JTLIST A2 . PROGN1) ; it is not the last element ;
(JUMP . EVAL) ; it is the last element ;

; LIST : FSUBR, EVLIS : 1SUBR ;

LIST
EVLIS (JFLIST A1 TST-) ; nothings to do ;
(CONS A2 NIL NIL) ; prepare the head of the result ;
(MPUSH A2) ; which is saved in the stack ;
; A2 is also the address of the
; last CONS-cell ;
EVLIS2 (UNCONS A1 A1 +TST) ; next element ;
(MOVC +TST A2 . EVAL) ; save the remainder and
; evaluate the element ;
(CONS A2 A1 NIL) ; CONS the value ;
(MPOP A3 A1) ; restore last and the remainder ;
(RPLACD A3 A2)
(JTLIST A1 . EVLIS1) ; list not exhausted ;
(CDRR A1 TST-)

; LESCAPE : FSUBR ;
; allows to force a tail recursion ;

LESCAPE (MOVJ +TST (*TR*) . PROGN)

; IF : FSUBR. The most simple conditionnal function ;
; allows to handle the tail-recursive functions ;

IF (UNCONS A1 A1 +TST)
(CALL . EVAL) ; evaluate the predicate ;
(UNCONS TST- A2 A3)
(JTNIL A1 . PROGNA3) ; else clauses ;
(MOVJ A1 A2 . EVAL) ; then clause ;

; COND : FSUBR. The most famous CONDITIONnal function ;
; allows to handle the tail-recursive functions ;

COND (MOVE A2 A1)
COND1 (JFLIST A2 TST-) ; no more clauses ;
(UNCONS A2 A1 +TST) ; A1 <- next clause ;
(UNCONS A1 A1 +TST) ; A1 <- the predicate ;
(CALL . EVAL) ; evaluate it ;
(MPOP A3 A2)
(JTNIL A1 . COND1) ; the predicate is false ;
(JFNIL A3 . PROGNA3) ; evaluate the clause ;
(RETURN) ; the clause is empty ;

; OR AND : FSUBR, logical connectors ;
; allows to handle the tail-recursive functions ;

OR (UNCONS A1 A1 A2)
(JFLIST A2 . EVAL) ; the last element ;
(MOVC +TST A2 . EVAL)
(JFNIL A1 . PRET)
(MOVJ A1 TST- . OR)

AND (JFLIST A1 . TRUE) ; (AND) -> T ;
AND1 (UNCONS A1 A1 A2)
(JFLIST A2 . EVAL) ; the last element ;
(MOVC +TST A2 . EVAL)
(JTNIL A1 . PRET)
(MOVJ A1 TST- . AND1)

PRET (MOVR A2 TST-) ; POP and return ;

; WHILE : FSUBR ;

WHILE (MOVJ +TST A1 . WHILE2) ; stack the whole expression ;
WHILE1 (CDR A1 TST)
(CALL . PROGN)
WHILE2 (MOVC A1 TST . EVALCAR) ; evaluate the test ;
(JFNIL A1 . WHILE1) ; it is ready for an other turn ;
(MOVR A2 TST-) ; finish ;

```

-----;  
; predicates and searches ;  
-----;

; NULL NOT ATOM NUMBF LISTP : 1SUBR ;

NULL (JTNIL A1 . TRUE)  
NOT (MOVR A1 NIL)  
ATOM (JFLIST A1 . TRUE)  
NUMBF (MOVR A1 NIL)  
LISTP (JTNUMB A1 . TRUE)  
(MOVR A1 NIL)  
(JTLIST A1 . TRUE)  
(MOVR A1 NIL)

; EQ NEQ : 2SUBR ;

EQ (JEQ A1 A2 . TRUE)  
NEQ (MOVR A1 NIL)  
(JNEQ A1 A2 . TRUE)  
(MOVR A1 NIL)

; EQUAL NEQUAL : 2SUBR ;

NEQUAL (MPUSH . NOT)  
EQUAL (MOVC A6 ST . EQUAL2)  
(MOVR A1 ('T))  
EQUAL1 (JFLIST A2 . NAN)  
(UNCONS A1 A1 +TST)  
(UNCONS A2 A2 +TST)  
(CALL . EQUAL2)  
(MPOP A2 A1)  
EQUAL2 (JTLIST A1 . EQUAL1)  
(JEQ A1 A2 TST-)  
NAN (MOVJ ST A6 . FALSE)

; prepare A6 for fast return ;

; cdr down A1 ;  
; cdr down A2 ;  
; recurse on CAR ;

; iterate on CDR ;

; fast return ;

TRUE (MOVR A1 ('T))  
FALSE (MOVR A1 NIL)

; CAR CDR : 1SUBR ;

CAR (CARR A1 A1)  
CDR (CDRR A1 A1)

; GET : 2SUBR ;

GET (GET A2 A1)  
(MOVR A1 AX)

; MEMQ : 2SUBR ;

MEMQ1 (JEQ (@ . A1) A2 TST-)  
(CDR A1 A1)  
MEMQ (JTLIST A1 . MEMQ1)  
(RETURN)

; the list is empty ;

-----;  
; create and modify ;  
-----;

; MAPC : 2SUBR, the position of the arguments is non-standard ;

MAPC (EXCH A1 A2)  
(JFLIST A1 TST-)  
MAPC1 (UNCONS A1 A1 +TST)  
(MOVC +TST A2 . APPLYC)  
(MPOP A2 A1)  
(JTLIST A1 . MAPC1)  
(RETURN)

; A1 <- list of arguments  
A2 <- function ;  
; nothing to do ;

```

; MAPCAR : 2SUBR ;

MAPCAR (EXCH A1 A2) ; A1 ← list of arguments,
                    ; A2 ← function ;
        (CONS A3 NIL NIL)
        (MOVJ +TST A3 . MAPCAR2)
MAPCAR1 (MPUSH A3)
        (UNCONS A1 A1 +TST) ; next argument ;
        (MOVC +TST A2 . APPLYC) ; save the function ;
        (CONS A3 A1 NIL)
        (MPOP A2 A1)
        (RPLACD TST- A3)
MAPCAR2 (JTLIST A1 . MAPCAR1)
        (CDRR A1 TST-)

; RPLACA RPLACD : 2SUBR ;

RPLACA (RPLACA A2 A1)
        (MOVR A1 A2)
RPLACD (RPLACD A2 A1)
        (MOVR A1 A2)

; SETQ : FSUBR ;

SETQ (UNCONS A1 +TST A2) ; stack the name ;
      (UNCONS A2 A1 +TST) ; stack the remainder ;
      (CALL . EVAL) ; evaluate the value ;
      (MPOP A2 A3)
      (RPLACA A3 A1) ; set the new value ;
      (JFLIST A2 TST-) ; no more couple ;
      (MOVJ A1 A2 . SETQ)

; SET : NSUBR, SETQQ : FSUBR ;

SETQQ
SET (UNCONS A1 A2 A1)
    (UNCONS A1 (@ . A2) A1)
    (JTLIST A1 . SET)
    (MOVR A1 (@ . A2))

; NEXTL : FSUBR ;

NEXTL (CAR A2 A1) ; A2 ← atom ;
      (CAR A3 A2) ; A3 ← its value ;
      (UNCONS A3 A1 A3)
      (RPLACAR A2 A3)

; CONS : 2SUBR ;

CONS (CONS A1 A2 A1)
      (RETURN)

; REVERSE : 2SUBR ;

REV1 (UNCONS A2 A3 A2)
      (CONS A1 A3 A1)
REVERSE (JTLIST A2 . REV1)
        (RETURN)

) ; end of the SETQ -INTERPRETER ; )

```

; initialisation of the indicators of the standard functions ;

```

(MAPC
 '(ATOM CAR CDR EPROGN EVAL EVLIS
  LISTP NULL NUMBP)
 '(LAMBDA (X) (PUT X 1 'TYPFN)))

(MAPC
 '(APPLY CONS EQ EQUAL GET MAPC MAPCAR
  MEMQ NEQ NEQUAL REVERSE RPLACA RPLACD)
 '(LAMBDA (X) (PUT X 2 'TYPFN)))

(MAPC
 '(PRINT PRIN1 TERPRI READ SET)
 '(LAMBDA (X) (PUT X 3 'TYPFN)))

(MAPC
 '(AND COND IF LESCAPE LIST NEXTL OR
  PROGN QUOTE SETQ SETQQ WHILE)
 '(LAMBDA (X) (PUT X 4 'TYPFN)))

```

'-INTERPRETER)