The words in a Minimal Thread Code Forth
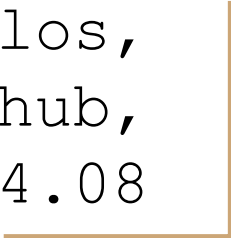
Alvaro G. S. Barcellos,
agsb at github,
version 1.0, 2024.08

the inner interpreter

`: NEXT IP )+ W MOV W )+ ) JMP ;`

" *Now Forth was complete. And I knew it.*"

Charles H. Moore,

"Forth - The Early Years", PDP-11

The inner interpreter is Forth's *heartbeat*.

The dictionary is the Forth's *DNA*.

Notes

In the following examples:

The names NEXT, NEST (aka DOCOL), UNNEST (aka SEMIS), are classic names for Forth inner interpreter.

Using reference ITC from FIG-Forth for PDP-11

Using PDP-11, Instruction Set Architecture (ISA)

By convention, stacks grows downward,

(X) indirect access, increment+ (pós)+, -decrement -(pré),

ISA order is: operator "from", "into"

The 'jump and link' concept was used by Charles H. Moore in 70's when --IP holds the address to next instruction--.

thread code models

STC: header | call word | sequence of "call words" | jmp EXIT

ITC: header | DOCOL | list of references to words | EXIT

DTC: header | call DOCOL | list of references to words | EXIT

MTC: header | list of references to words | EXIT

STC subroutine thread code, ITC indirect thread code

DTC direct thread code, MTC minimal thread code

DOCOL is NEST and EXIT is UNNEST

the Minimal (and direct) Thread Code

1. *Any primitive word contains only native machine code*

2. *Any compound word contains only a list of word references*

3. *All compound words have the references 'pushed and pulled' at the return stack*

4. *Any compound word should only access the return stack using >R R> R@*

5. *All primitives words must be before the compounds words in the dictionary*

the MTC dictionary

```
+------------+ start of memory
| internals  |   system and support
+------------+
| primitives |   only "native code" words
+------------+ init
| compounds  |   only "list of references" words
+------------+ here
|            |
|            |   free memory
|            |
+------------+ end of memory
```

organize the dictionary:
all primitives words
must be before
the compounds words
in the dictionary

the MTC dictionary

primitive words: only native code

```
+-------+------+------+------+----------+
| code | code | code | code | jump exit |
+CFA---+------+------+------+----------+
```

compound words: only list references

```
+-------+--------+-------+--------+-------+
| 'HERE | 'STORE | 'CELL | 'ALLOT | 'EXIT |
+CFA----+--------+-------+--------+-------+
```

PS. Not showing the headers
    No need "DOCOL" at CFA of compound words !

the PDP-11 code of MTC

```
unnest:  MOV (RP)+, IP      // .pull

next:    MOV (IP)+, W

pick:    CMP W, _init_      // .pick

         BMI jump

nest:    MOV IP, -(RP)      // .push

         MOV W, IP

         JMP next

jump:    JMP (W)
```

```
from ITC FIG-Forth, 1.3.3.1:

SEMIS :  MOV (RP)+, IP

NEXT :   MOV (IP)+, W
         JMP @(W)+

DOCOL :  MOV IP, -(RP)
         MOV W, IP
         JMP NEXT
```

init is the starting address of the compound words at dictionary

ITC, DTC and MTC

In ITC and DTC

1. in compound words, CFA must reference the DOCOL routine
   (or DOCON, DOVAR, DOLIT, DODOES, DODEFER, etc)
2. must jump twice for every compound word
   NEXT -> WORD -> DOCOL -> NEXT

In MTC

1. in compound words, CFA could reference any word
2. test all word references and jump ONLY when it is
   bellow the "init" address, the starting of compound
   words in dictionary

inside words

MTC, PDP-11

```
unnest:  MOV (RP)+, IP      // .pull

next:    MOV (IP)+, W

pick:    CMP W, init

         BMI jump

nest:    MOV IP, -(RP)      // .push

         MOV W, IP

         JMP next

jump:    JMP (W)            // .jump
```

pull, push, pick, jump

syntax: operator from, into

wp, word pointer, scratch
ip, link pointer, reserved
rp, return stack pointer, reserved

MTC, Risc-V

```
unnest:  lw IP, 0(RP)
         addi RP, RP, 1 * #CELL
next:    lw W, 0(IP)
         addi IP, IP, 1 * #CELL
pick:    BMI W, _init_, jump
nest:    addi RP, RP, -1 * CELL
         sw 0(RP), IP
         add IP, W, zero
         jal zero, next
jump:    jalr zero, 0(W)
```

pull, push, pick, jump

syntax: operator into, from

do not use default link register

let assembler decide offsets for
                (jalr zero, zero, link)

MTC, 6502

An implementation of Minimal Thread Code (and Direct Thread Code) for 6502 CPU is at the milliforth-6502,
https://github.com/agsb/milliforth-6502

The scope of milliforth-6502 is to make a near boot size Forth as sectorForth and milliForth, both for x86.

The original milliForth: https://github.com/fuzzballcat/milliForth

The inspirational sectorForth:
https://github.com/cesarblum/sectorforth/

```
header "exit","exit"

_unnest: // .pull
    lw s6, 0(s5)
    addi s5, s5, 1*CELL
    // jal zero, _next

_next:   // .next
    lw s9, 0(s6)
    addi s6, s6, CELL
_pick:   // .pick
    bmi s9, _init_, _jump
    // jal zero, _nest

_nest:   // .push
    addi s5, s5, -1*CELL
    sw s6, 0(s5)
    add s6, s9, zero
    jal zero, _next
```

```
_jump: // .jump
    // to insert debug info
    jalr zero, s9, 0

_link: // .link
    // to insert debug info
    jal zero, _unnest

// not optimized :)
```

A inner
example
with
RISC-V

RISC-V, R32i, word is 32 bit cell, inner interpreter

Minimum Thread Code
s5, return stack (RS), ~ reserved
s6, next reference (IP), ~ free for use outside
s9, word reference (Wd), ~ free for use outside
zero, r0 register, is always zero, hardware wired
CELL is cell size in bytes
less than 12 instructions for Forth inner interpreter

To allow further words been defined using native code, as like primitives, is need a primitive word for executing it, by doing a immediate jump to following address, bypassing the inner interpreter.

(in FigForth done by SEMIC and PSCODE)

LEAP ? LOPE ? DIVE ? maybe CODE: and ;CODE ?

```
NEXT      (ITC, EXECUTE FORTH WORD WHOSE CODE ADDRESS IS ON IP)

    MOV (IP)+,W

    JMP @(W)+

EXEC      (ITC, EXECUTE FORTH WORD WHOSE CODE ADDRESS IS ON STACK)

    MOV (S)+,W

    JMP @(W)+

LEAP      (MTC, EXECUTE NATIVE CODE WHOSE CODE ADDRESS IS IN IP)

    JMP (IP)

                              (NEXT and EXEC depends on DOCOL)
```

Easy summary

In ITC and DTC, the inner interpreter

Always jumps to address at first cell (CFA) of word definition,

In compounds words, the CFA must be DOCOL (or DOVAR, DOCON, DOLIT, etc)

In MTC, the inner interpreter

Only jumps when CFA is a reference to primitive word.

All primitives words must be before the compounds words in the dictionary,

In compounds words, the CFA could be any word.

Uses less memory in dictionary, without DOCOL (&etc) at each word

conclusion

When reinvent the wheel ?

MTC is a faster inner interpreter;

MTC needs little changes at thread code;

MTC uses less memory and less jumps;

MTC shortens the dictionary;

MTC effective as DTC and ITC;

Why PDP-11 ?

the PiDP-11 is a replica of the PDP-11/70, with a Raspberry Pi running the simh simulator for PDP-11/70, in a Linux Debian.

```
: NEXT IP )+ W MOV W )+ ) JMP ;

   IP )+ W  MOV is W = (IP), IP++

   W )+ ) JMP      is PC = W, W++
```

https://github.com/agsb

references

https://library.nrao.edu/public/memos/comp/CDIR_17.pdf

https://pdos.csail.mit.edu/6.828/2005/readings/pdp11-40.pdf

http://www.stackosaurus.com/figforth-1.3.3.1/FORTH.MAC

http://www.complang.tuwien.ac.at/forth/threaded-code.html

http://www.bradrodriguez.com/papers/moving1.htm

https://muforth.nimblemachines.com/threaded-code/

http://git.annexia.org/?p=jonesforth.git;a=tree

https://home.hccnet.nl/a.w.m.van.der.horst/lina.html

https://github.com/simh/simh

myself

Myself,

Geologist, Master in Geophysics,

I use computers for data acquisition and calculations,

and an electronics and circuits hobbyist.

I discovered Forth in 1980, in Byte Magazine, in the UFRJ-BR library,

and rediscovered Forth in 2019, with the Forth2020 forum.

My projects at https://github.com/agsb

Primitive Sequences

docol	call
docon	lit @
dovar	lit
douser	useraddr
dodefer	lit @ exec
dofield	lit +
dodoes	lit call

https://www.academia.edu/105385974/Primitive_sequences_in_general_purpose_Forth_programs

call or link

from PDP-11 ISA

```
        JSR Ri, Src  // Jump into subroutine
            -(SP) ← Ri; Ri ← +PC; PC ← Src;
        RTS           // Return from subroutine
            PC ← Ri; Ri ← (SP)+;

push and pull    // same, but do not use Ri
        CALL Src
            -(SP) ← Ri; Ri ← +PC; PC ← Src;
        RETURN
            PC ← Ri; Ri ← (SP)+;

link and jump    // same, but do not use SP
        JAL Ri, Src
            -(SP) ← Ri; Ri ← +PC; PC ← Src;
        JR Ri
            PC ← Ri; Ri ← (SP)+;
```