




Minimal Indirect Thread Code Forth

Alvaro Barcellos,
agsb at github,
version 0.93, 2022



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

Ideally, there is two types of words:

primitive, that contains only machine code without calls

compound, that contains only references of words

first-word: DOCODE for primitive and DOCOLON for compound

last-word: EXIT for primitive and SEMIS for compound

parameters: could be pure code or a list of references

a primitive is also called leaf, a compound is also called twig

the classic indirect thread code

compound: ; a NEST and a UNNEST

LINK	6	D	O	U	B	L	E	0	NEST	DUP	PLUS	UNNEST
------	---	---	---	---	---	---	---	---	------	-----	------	--------

primitive: ; a DOCODE (jump to *self*) and a *EXIT* (macro jmp NEXT)

LINK	3	D	U	P	DOCODE	code	code	code	EXIT
------	---	---	---	---	--------	------	------	------	------

the ITC code

NEXT:

Fetch the address pointed by IP onto WR

Increment IP by address size

Jump to address at WR

NEST: (aka DOCOLON, at start of words)

Push IP onto call stack

Copy WR to IP

Execute *NEXT*

UNNEST: (aka SEMMIS, at end of words)

Pull IP from call stack

Execute *NEXT*

EXIT: (at end of code)

Execute *NEXT*

PS: non optimized pseudo code

ITC details

All compound words, does a call with return and two jumps.

All primitive words does three jumps.

Any call or jump could be a pipeline refill.

(Also in optimized codes, NEXT is placed between UNNEST and NEST, and executed two times.)

A proposal of
minimal indirect
thread code

the minimal indirect thread code

compound: ; a *UNNEST*, (where did *NEST* go ?)

```
+-----+---+-----+---+-----+---+-----+---+-----+---+-----+
| LINK  | 6 | D | O | U | B | L | E | 0 | DUP | PLUS | UNNEST |
+-----+---+-----+---+-----+---+-----+---+-----+---+-----+
+-----+---+-----+---+-----+---+-----+---+-----+---+-----+
```

primitive: ; a *NULL* and a jump, (where did self reference go ?)

```
+-----+---+-----+---+-----+---+-----+---+-----+---+-----+
+
| LINK  | 3 | D | U | P | NULL | code | code | code | LINK |
+-----+---+-----+---+-----+---+-----+---+-----+---+-----+
+
+-----+---+-----+---+-----+---+-----+---+-----+---+-----+
```

PS. NULL is 0x0, UNNEST is a primitive, LINK is 'jmp _unnest'

the MITC code

NEXT:

Fetch the address pointed by IP onto WR
Increment IP by address size
if WR is NULL, then Execute JUMP
else Execute NEST

NEST: (aka DOCOLON)

Push IP onto call stack
Copy WR to IP
Execute *NEXT*

UNNEST: (aka DOSEMIS)

Pull IP from call stack
Execute *NEXT*

PS: non optimized pseudo code

the MITC code

JUMP:

Jump to address in IP

LINK: (*Link is same as Exit*)

Execute UNNEST

; both could also be hooks for debug

PS: non optimized pseudo code

Details

The above code only performs a jump to a primitive word, whose header is NULL (0x0).

Does only a compare per Forth word, to decide if executes a NEST or a JUMP.

All compound word references are placed and removed, onto the return stack, do not executing any jump.

All references are passed by return stack.

Uses jump and link concept, as modern RISC-V processors does.

Conclusions

The functionality of the classic ITC implementation is maintained.

Only primitive words are dependent on the processor-specific instruction set.

The interpreter becomes more effective, because it only performs jumps to the primitive words.

The dictionary becomes more compact as it uses one less reference in all compound words.

Ideas

In microcontrollers with Harvard architecture, a vocabulary with primitives words (machine code) stay in rom/flash, and vocabularies with compound words (pure references) stay in sram.

Could use a "trampoline" function for jump to a list of primitives, as "token table".

Vocabularies with compound words could be compiled and shared as relocatable libraries.

PS. the examples names NEXT, NEST, UNNEST, EXIT are classics from Dr. Chen-Hanson Ting, for eForth inner interpreter. JUMP and LINK are traditional.

```

header "UNNEST", "UNNEST"
    .word 0x0 ; nop

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

_next: // cast
    lw s9, 0(s6)
    addi s6, s6, CELL
    beq s9, zero, _jump
    // jal zero, _next

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

```

```

_jump: // pass
    // insert debug info
    jalr zero, s6, 0

_link: // next
    // insert debug info
    jal zero, _unnest

// not optimized :)

```

example
with
RISC-V

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

Extended Indirect Thread Code

s5, return stack (RS), ~reserved
 s6, next reference (IP), ~ free for use outside
 s9, temporary (WK), ~ free for use outside
 zero, r0 register, is always zero, hardware wired
 CELL is cell size in bytes
 only 12 instructions for Forth inner interpreter

Small side effect

Small side effect, transparent inside implementation:

>R must put value into second position on return stack

R> must get value from second position on return stack

R@ must copy the value at second position on return stack

(Because first position is itself the return reference)

All words must only use >R R> R@ to access the return stack

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

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

W)+) JMP is jump (W), W++

<https://github.com/agseb/f2u>

<https://github.com/agseb/immu>

https://en.wikipedia.org/wiki/Threaded_code

digital

pdp11/70

RSA
ERR

ADDR
ERR

RUN

PAUSE

MASTER

USER

SUPER

KERNEL

DATA

ADDRESSING
16 22

ADDRESS

POWER
LOCK

OFF

PARITY
HIGH LOW

DATA

USER D USER I

SUPER D SUPER I

KERNEL D KERNEL I

CONS PHY PROG PHY

DATA
PATHS

BUS REQ

II ADR
TRP CPU

DISPLAY
REGISTER

21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

LONG
ADDR

EXAM

IMP

CONT

HALT

ENABLE
S. INT

5. INT

STOP