this presentation is out-dated and preserved only for reviews.

a new version of MITC with better code for

link and jump, and, push and pull,

at https://github.com/agsb/immu .

Minimal Indirect Thread Code Forth

Alvaro G. S. Barcellos,
agsb at github,
version 0.9, 2023

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*.

the dictionary

Ideally, there are two types of words (thread code):

primitive, that contains only machine code without calls

compound, that contains only references to words

and all words have header, first-word, parameters, last-word.

first-word: *DOCODE* for primitive and *DOCOLON* for compound

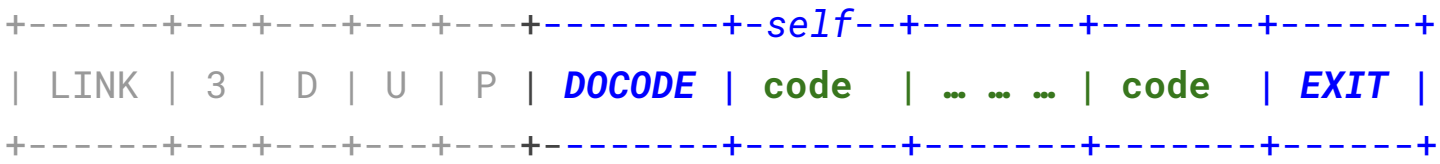
parameters: could be pure code or a list of references

last-word: *EXIT* for primitive and *SEMIS* for compound

the classic indirect thread code ITC

compound: ; a NEST and a UNNEST

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



- aka, NEST is DOCOL and UNNEST is SEMIS

the ITC code

```
NEXT:
     Fetch the address pointed by IP onto WR
     Increment IP by address (cell) 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 push and three jumps. (A call is a push and a jmp)

All primitive words, does three jumps.

Any call or jump could cause a pipeline refill.

Also, in optimized implementations, NEXT is placed between UNNEST and NEST, and executed two times.

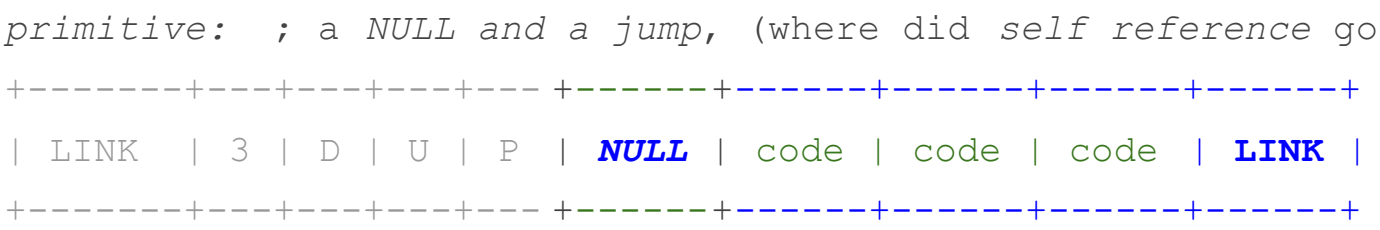
Using the term ‘Execute’ because could be implemented as a inline macro or a ‘Jump to’.

A proposal of minimal indirect thread code

the minimal indirect thread code

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

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



PS. NULL is 0x000H, UNNEST is a primitive, LINK is ‘jump _unnest’

the MITC code

```
NEXT:
     Fetch the address pointed by IP onto WR
     Increment IP by address (cell) 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
; IP and WR are free for any use
;
```

PS: non optimized pseudo code

Details

The above code only performs a jump to a primitive word, whose header is NULL (0x0000).

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, and which C. H. Moore used in 60's

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 address for those)

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

No need to keep IP or WR, all references are into return stack

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

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

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

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

```
_jump: // .cast
     // to insert debug info
     jalr zero, s6, 0

_link: // .pass
     // 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

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

Conclusions

Using MITC,

The functionality of the classic ITC implementation is maintained.

The interpreter becomes more effective, because it only performs ‘jump and link’ to the primitive words.

The dictionary becomes more compact, it uses one less reference in all the 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”. Use the number of primitives instead NULL for check for a primitive.

Vocabularies with compound words could be compiled and shared as relocatable libraries, because references ‘tokens’ to primitives words.

Small Notes

"Thou shalt not PICK; and Thou shalt never ROLL."

In the examples, the names NEXT, NEST, UNNEST, EXIT are classic names for Forth inner interpreter. JUMP and LINK are traditional.

In the dictionary view as a tree, a primitive word is a leaf and a compound word is a branch.

Fig-Forth, cmForth and e-Forth implementations are used as public resources.

Prefer do not mix references and code inside a word

A compound word is *where two or more words merge to form a new one.*

Some references

http://www.forth.org/fig-forth/fig-forth_6502.pdf

http://www.forth.org/fig-forth/fig-forth_PDP-11.pdf

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

http://www.forth.org/OffeteStore/1010_SystemsGuideToFigForth.pdf

http://www.forth.org/OffeteStore/1013_eForthAndZen.pdf

http://www.forth.org/OffeteStore/1003_InsideF83.pdf

https://muforth.nimblemachines.com/

https://muforth.nimblemachines.com/call-versus-branch-and-link/

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

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

W)+) JMP is PC = W; W++

https://github.com/agsb/immu

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


digital
pdp11/70
RUN
PAUSE
MASTER
USER
SUPER
KERNEL
DATA
ADDRESSING
ADDRESS
DATA
PARITY
HIGH
LOW
POWER
LOCK
OFF
USER D
USER I
SUPER D
SUPER I
KERNEL D
KERNEL I
CONS PHY
PROG PHY
DATA PATHS
BUS REG
DISPLAY REGISTER
21
20
19
18
17
16
15
14
13
12
11
10
9
8
7
6
5
4
3
2
1
0