ASE: Writing a forth interpreter from scratch

Pablo de Oliveira <pablo.oliveira@uvsq.fr>

January 18, 2013
Section 1

Introduction
Why an embedded Forth interpreter?

- Forth is minimal: writing a Forth interpreter for a new architecture is simple and fast.
  - A full Forth system can be written in less than 2000 lines of codes.

- Forth is powerful for testing embedded systems:
  - Comes with a REPL (Read-Eval Print Loop), we can test the target interactively.
  - It is very easy to define new words to control the target.

```forth
LEFT-MOTOR 50 SPEED
2 LED ON

: TURN-RIGHT ( -- )
  RIGHT-MOTOR 0 SPEED
  LEFT-MOTOR 50 SPEED
  2 WAIT
  LEFT-MOTOR 0 SPEED
;
```
Lecture Goal: Building a forth interpreter from scratch!

- Know how to build Forth from scratch starting from assembly.
- We study Richard W.M. Jones's Forth minimal implementation. Most of the code samples in this lecture are borrowed from Jones's Forth. http://git.annexia.org/?p=jonesforth.git
- Target: x86 architecture, you will port it to ARM!
Section 2

The execution model
Execution Model

- In a forth system there are two kind of words definitions:
  - Native words: these words are written in assembly (or other low level language).
  - Forth words: these words are written in forth by calling other native or forth words.
- Our execution model needs to be able to execute both kind of words.
Call Threaded Code

: SQUARE DUP * ;

SQUARE: ( a forth word )
    call DUP
    call MUL
    ret

DUP: ( a native word )
    mov (%esp), %eax
    push %eax
    ret

MUL: ( a native word )
    pop %eax
    pop %ebx
    imull %ebx, %eax
    push %eax
    ret

▷ Simple but overhead of call and ret instructions.
Direct Threaded Code

Instead of the calls, we store the addresses of the words:

: SQUARE DUP * ;

SQUARE:
  &DUP
  &MUL <-- %esi points to the next word to execute
  &EXIT

A definition is a list of addresses and not executable. We introduce a new assembly macro NEXT. NEXT is called at the end of each word execution. It jumps to the next word (pointed by %esi) and increments %esi.

NEXT:
  lodsl // loads (%esi) into eax and increments %esi
  jmp *%eax
Direct Threaded Code

SQUARE:
&DUP
&MUL
&EXIT

DUP:
    mov (%esp), %eax
    push %eax
    NEXT
MUL:
    pop %eax
    pop %ebx
    imull %ebx, %eax
    push %eax
    push %eax
    NEXT

Something is missing:

- How do we start executing SQUARE?
- How do we call SQUARE from another word?
Direct Threaded Code

SQUARE:

CALL DOCOL<-.  
&DUP       |
&MUL       |
&EXIT      | EXIT:
&EXIT      | NEXT:     mov (%ebp), %esi
POW4:       | lodsl    add $4, %ebp // Restore old IP
CALL DOCOL  | jmp *%eax NEXT
&SQUARE ----'
&SQUARE
&EXIT

DOCOL:

sub $4, %ebp
mov %esi, (%ebp) // Save the old IP on the stack
add $4, %eax     // %eax points to the address of SQUARE DOCOL
                 // We increment it to point to &DUP
mov %eax, %esi  
NEXT
Indirect Threaded Code

- Direct Threaded Code
  - Overhead of one call at the start of each Forth word.
  - Cache usage is non-optimal because we mix data and code.
  - Still very fast and simple.

- Indirect Threaded Code
  - We add one level of indirection:

    We replace:

    \[
    \begin{align*}
    \text{SQUARE:} & \quad \text{SQUARE:} \\
    \text{CALL DOCOL} & \quad \text{&DOCOL} \\
    \text{&DUP} & \quad \text{with} \quad \text{&DUP} \\
    \text{&MUL} & \quad \text{&MUL} \\
    \text{&EXIT} & \quad \text{&EXIT}
    \end{align*}
    \]

  - Reduces a bit the code size at the cost of an indirection.
  - Does not mix code and data.
Execution Model Conclusion

- The execution model specifies how forth words are executed.
- Jones’s Forth uses Indirect Threaded Code as most forths.
- ITC works exactly as DTC but with an extra level of indirection:

  NEXT (DTC):
  
  lodsl // loads %esi into eax and increments %esi
  jmp *%eax

  |   |   |
  |   |   |
  V

  NEXT (ITC):
  
  lodsl // loads %esi into eax and increments %esi
  jmp *(%eax)
Section 3

Literals
How to add data inside a forth word?

: DOUBLE (n -- n) 2 * ;

is compiled to

DOUBLE:
  &DOCOL
  2 <- This is not an address. NEXT will fail.
  &MUL
  &EXIT

Idea: use special word LIT. LIT will push 2 in the stack and skip 2.

DOUBLE:
  &DOCOL
  &LIT
  2
  &MUL
  &EXIT
DOUBLER:
    &DOCOL
    &LIT
    2
    &MUL
    &EXIT

How is LIT implemented?

LIT:
    lodsl  // read literal (pointed by %esi) into %eax
    push %eax  // push literal into the stack
    NEXT
Section 4

Dictionary
The Dictionary

- In Forth words are kept into a Dictionary.
- It is a linked list:

    ```forth
    NULL
    ^
    | (4b) (1b) ..... (4b aligned)
    +--|-----------------+---+---+---+---+---+---+---+---+------------- - - - -
    | LINK | 6 | S | Q | U | A | R | E | 0 | (definition ...)
    +-----------------+---+---+---+---+---+---+---+---+------------- - - - -
    ^             len padding
    |
    +-----------------+---+---+---+---+---+---+---+---+------------- - - - -
    | LINK | 4 | P | O | W | 4 | 0 | 0 | 0 | (definition ...)
    +-----------------+---+---+---+---+---+---+---+---+------------- - - - -
    ^             len padding
    |
    LATEST
    ```
Forth words : SQUARE

+-----------------+---+---+---+---+---+---+---+---+-------+-----+---+------+
| LINK | 6 | S | Q | U | A | R | E | 0 | DOCOL | DUP | * | EXIT |
+-----------------+---+---+---+---+---+---+---+---+-------+-----+---+------+

len name pad
Native (assembly) words : DUP

+------------------------------------------+
| LINK | 3 | D | U | P | CODEOFDUP |
|------------------------------------------|

len name

CODEOFDUP:
  mov (%esp), %eax
  push %eax
  NEXT
How to get the code address of an entry?

- To get the code address of an entry we use the CFA word.

```
+-----------------+
| LINK  | 3 | D | U | P | CODEOFDUP |
+-----------------+
  len  name ^
  |        |
  |        |
  '-----------------'

>CFA
```

The implementation of CFA is simple, the only complication is calculating the padding size to skip. Left as an exercise for the reader!
How to find an entry?

- FIND (name? – address).
- FIND start at latest, and traverses the linked list.
- For each entry it compares the name of the entry with name?. If they match, FIND returns the address of the entry.
- The code is simple.

```
pop %ecx ; pop %edi // %ecx = length, %edi = address
push %esi           // save %esi which is used by cmpsb

mov LATEST,%edx    // LATEST points to latest word
1: test %edx,%edx   // NULL pointer? (end of the linked list)
   je 4f            // Word not found return NULL

// Compare the length
   xor %eax,%eax
   movb 4(%edx),%al // length field
   cmpb %cl,%al     // Length is the same?
   jne 2f           // Not the same
```
How to find an entry?

push %ecx          // Save the length
push %edi          // Save the address (repe cmpsb will move this)
lea 5(%edx),%esi   // Dictionary string we are checking against.
repe cmpsb         // Compare the strings.
pop %edi
pop %ecx
jne 2f             // Not the same.
// The strings are the same – return the header pointer in %eax
    mov %edx, %eax
    pop %esi
    ret

2: mov (%edx),%edx  // Move to the previous word
    jmp 1b           // .. and loop.
Section 5

Native Words
Before writing forth words in forth we need to add a set of primitive native words.

- DUP, DROP, SWAP, OVER, ROT, +, *, /MOD, =, <, 0=, etc...

Jones’s forth uses an assembly macro to add words to the dictionary:
- The macro adds a link to the address of the previous word (LINK).
- It updates LINK with the new word’s address.
- It adds the len and name field.

```assembly
defcode "DUP",3,,DUP
    mov (%esp),%eax // Read top of the stack in %eax
    push %eax       // Push %eax on the stack
    NEXT
```
Adding native words to our forth

EXERCICE: Give assembly implementation of

- DROP: drops the first element of the stack.
- OVER: reads the second element of the stack and pushes it to the top.
- +: adds the top two elements of the stack.
- ! (data address –): write data at address
- @ (address – data): reads data at address
Adding native words to our forth

defcode "DROP",4,,DROP
    pop %eax
    NEXT
defcode "OVER",5,,OVER
    mov 4(%esp), %eax
    push %eax
    NEXT
defcode "+",1,,ADD
    pop %eax
    add %eax, (%esp)
    NEXT
defcode "!",1,,STORE  defcode ":",1,,FETCH
    pop %ebx // address
    pop %eax // data
    mov %eax, (%ebx)
    push %eax
    NEXT
    pop %ebx // address
    mov (%ebx), %eax
    push %eax
    NEXT
Section 6

10
Input Output

- **KEY (−c)**: Reads a character from stdin.
- **EMIT (c−)**: Writes a character to stdout.
- **WORD (−addr length)**: Reads the next word from stdin and stores it into the stack as (address, length)
- **NUMBER (−n)**: Reads a number from stdin.
- In Jones’s forth these are implemented in assembly (< 100 lines). We do not discuss their implementation here, but feel free to check it out!
Section 7

Branching
Branching BRANCH, 0BRANCH

BRANCH and 0BRANCH are like LIT, they are followed by a NUMBER. In this case, the number represents a jump offset.

- BRANCH OFFSET (-) : Increments the IP
- 0BRANCH OFFSET ( cond – ) : If cond is 0, increment

```assembly
defcode "BRANCH",6,,BRANCH
    add (%esi), %esi
    NEXT

defcode "0BRANCH",7,,ZEROBRANCH
    pop %eax // Read cond
    test %eax, %eax
    jz BRANCH
    lodsl // Otherwise skip the offset
    NEXT
```
Summary until now

- First, we decided to use Indirect threaded code. We implemented NEXT, DOCOL and EXIT.
- Next, we implemented LIT to mix code and data in a word definition.
- Then, we defined the dictionary structure and added Native assembly words.
- Until now everything is hardcoded. Now we get into compiling new words!
Section 8

Compiling new words
Writing to memory: COMMA

, is a forth word that stores the top of the stack at HERE and increments HERE.

defcode ",",1,,COMMA
    pop %eax // Get the top of the stack
    mov HERE, %edi // Load HERE address in %edi
    stosl // Store the top of the stak in %edi
    mov %edi, HERE // Update HERE address
    NEXT
CREATE takes a string name on the stack and creates a new dictionary entry on the user memory.

```assembly
defcode "CREATE", 6, ,CREATE
    pop %ecx ; pop %ebx // Read the length and address of
        // the string name.
    mov HERE, %edi // HERE points to the first free address
        // in user memory
    mov LATEST, %eax // LATEST points to the last defined word
    stosl // Store the link

    mov %cl, %al // Read the length
    stosb // Store the length
```
push %esi  // Save %esi
mov %ebx, %esi  // Put the address of the name in %esi
rep movsb  // Store the name
pop %esi  // Restore %esi
add $3, %edi
and $~3, %edi  // Compute padding size

mov HERE, %eax  // Update variables
mov %eax, LATEST
mov %edi, HERE
NEXT
Compile and Immediate mode

- The forth interpreter usually is in immediate mode. It reads words from stdin and executes them.
- We can use a special word `[` to get into compile mode. In compile mode the interpreter reads words from stdin but writes their address to HERE.
- To get out of compile mode, we use `]`.
- Some words are flagged as IMMEDIATE. IMMEDIATE words are always executed, both in compile and immediate modes.

The current mode is stored in a global variable STATE

```
defcode "[",1,F_IMMED,LBRAC
    mov $0, STATE
    NEXT

defcode "]",1,F_IMMED,RBRAC
    mov $1, STATE
    NEXT
```
Getting the address of words, TICK


“”’ word gets the address of the next word on the stack. So for example ’ SQUARE will return the CFA of SQUARE.

def code "''",1,,TICK
   WORD
   FIND
   >CFA
   NEXT
Compiling new words, COLON

Now everything is ready to define “:"  

COLON:  
    WORD ( Read the next word into the stack as a string )  
    CREATE ( Create a new dictionary entry named after the string )  
    ' DOCOL ' ( Compile the address of DOCOL )  
    [ ( Enter compilation mode )  
    EXIT
And to end the compilation of a new word we use ";’’

```
SEMICOLON: IMMEDIATE
’ EXIT , ( Compile the address of EXIT at the end )
] ( Exit compilation mode )
EXIT
```

Why must “;” be IMMEDIATE?
The interpreter

INTERPRET : ( in pseudo-code )
  WORD ( Read a word from stdin )
  FIND ( Find it in the dictionary )
  IF FOUND
    >CFA ( Get its code address )
    IF IMMEDIATE? or IMMEDIATE MODE
      JMP ( Jump to the code address )
    ELSE
      , ( Compile the code address to HERE )
  ELSE ( Not a word in dictionary )
    IF NUMBER? ( If it is a number )
      NUMBER ( Read the number )
      IF IMMEDIATE MODE
        PUSH NUMBER
      ELSE
        , ' LIT , , ( Compile LIT number )
    ELSE ERROR
Section 9

The rest
What about the rest?

- So, what about the rest? Where is NEGATE, IF, CONSTANT, VARIABLE, BEGIN UNTIL, and all the other forth words?
- Now that we bootstrapped the compiler, everything else can be written in forth!
NEGATE

: NEGATE ( n -- -n ) 0 SWAP - ;
CONSTANT is a forth word that creates a new word, here TEN, that pushes 10 on the stack.

10 CONSTANT TEN
TEN . CR
10

How can we define CONSTANT in forth?
CONSTANT

: CONSTANT ( n -- )
  WORD ( Read the name )
  CREATE ( Create a new dictionary entry )
    ' DOCOL , ( Compile DOCOL )
    ' LIT , ( Compile LIT )
    , ( Compile n )
    ' EXIT , ( Compile EXIT )

;

Calling 10 CONSTANT TEN will compile the following entry:
+-----------------------------------------------------------------------------+
| LINK  | 3 | T | E | N | DOCOL | LIT | 10 | EXIT |
+-----------------------------------------------------------------------------+
BEGIN UNTIL

: count ( n -- ) BEGIN 1- DUP . DUP 0= UNTIL ;
10 count 9 8 7 6 5 4 3 2 1 0

How to define BEGIN and UNTIL?
BEGIN UNTIL

: BEGIN IMMEDIATE
  HERE @  ( save location on the stack )
;

: UNTIL IMMEDIATE
  ' OBRANCH , ( Compile a conditional branch )
  HERE @ -  ( Compute offset )
  ,       ( Compile the offset )
;