The Heart of Forth

Forth can be seen as assembly language for a processor which has two stacks, the data stack and the return stack. Many processors have only one hardware stack, so the other one has to be simulated in software.

Comments, queries and suggestions gratefully received

Native Code (aka Subroutine Threading)

If the hardware return stack is used, then there need be no difference between primitive and secondary definitions - it is all native machine code. A Forth secondary definition

```
 : EASYAS  A B C ;
```

compiles to `CALL A CALL B CALL C` no matter what parameters are required by A B or C. Each word takes its own parameters from the data stack, and places its results there. It is up to the programmer, or the calling word, to ensure that the correct parameters are provided.

For speed, the bodies of short primitive definitions can be compiled in-line instead of compiling a call. There are many other optimisations that can be made - of increasing complexity and decreasing utility - but that is a subject I am not qualified to discuss. This method of implementation is increasingly popular, especially where Forths are written in C.

The Classic Approach - Threaded Code

Traditionally a secondary definition such as EASYAS contains a *code field* and a *data field* containing a list of execution tokens (xts). Each xt in the list is the address of the code field of one of the words in the definition. The advantages are compactness and simplicity. It can also be faster than unoptimised native code. There are two main varieties:

Direct Threaded
: Each code field contains a machine code fragment. So, in the case of a primitive (machine code) definition, the xt is the address of the machine code itself

Indirect Threaded
: Each code field contains a pointer to a native code fragment. With indirect code, each definition, of whatever type, has a code field exactly one cell wide, and a data field. In the case of a primitive definition, the data field contains its machine code, and the code field is simply a pointer to it. All types of word have the same format: the code field defines what the word does - the data field what it does it with. A word's xt is the address of its code field.

Direct threaded code is usually faster than indirect threaded, except on chips such as the Pentium, which dislike mixing code and data. For these threading mechanisms to work, each native code fragment must end in an instruction (commonly called NEXT) which advances execution to the next fragment.

The Virtual Chip

I am describing operations such as NEXT by use of a pseudo-assembler which uses Forthlike postscript notation. This allows me to show what is happening without reference to extra hypothetical registers. I am using the following conventions:

- The data stack, the return stack, and xts all have the same width - one *cell*, typically 16 or 32 bits. (This is not strictly necessary - so long as an xt will fit on both the data and return stacks, the data

stack may be wider to allow it to address more space - but it makes for a simpler model)
- *register* means the value held in that register
- [*xxxx*] means the value held at address xxxx
- *xxxx* TO *register* means store xxxx in the register
- *xxxx* +TO *register* means increment by that amount
- *xxxx* PUSH *register* the register acts as a stack, push xxx onto it
- POP *register* means the value popped from the stack

The Registers

Here are the registers of the virtual chip, listed in order of importance as regards speed:

PSP The Parameter Stack Pointer
: Gives access to the data stack. You need a minimum of 32 cells for the stack

IP The Instruction Pointer
: Points to the next exection token to be executed

RSP The Return Stack Pointer
: You need a minimum of 24 cells for the return stack. If you can, allot at least 64 cells for each stack and use the fastest memory you have.

W The Working Register
: Provides an address from which the data field of the currently executing word can be found. A primitive may overwrite W with impunity, but should preserve all the other virtual registers. My pseudocode assumes the address held in W to be the current xt, and that the data field immediately follows the code field, but neither of these are requirements. I use the pseudo-op DATA to hide the details how the data field address is obtained.

TOS Top of Stack Cache
: If you can spare a register (hereafter called TOS), cacheing the top value on the stack makes many primitive definitions faster. Using TOS can save a PUSH PSP and a POP PSP on definitions that use the stack but leave it balanced, at the expense of an extra register move when adding items. It's generally not worthwhile trying to cache more than one stack item. For clarity, I will not be using TOS in my pseudocode of the inner interpreter.

UP The User Pointer
: Only needed on multi-tasking Forths, it holds the base address for variables local to each task

Only a fraction of the available native opcodes are needed to build a Forth system, and it is a simple task, once you have Forth, to write an assembler that will implement it for another system.

The best threading method and register assignments for a particular system can only be decided by coding some primitives and timing them. The most frequently used ones are:

`NEXT, ENTER, EXIT, DOVAR, DOCON, LIT,& @, !, +, BRANCH, ?BRANCH, SWAP, >R, R>.`

For a more technical approach, with examples for several chips, see Brad Rodriguez's Moving Forth. The Forth model discussed there is based to a large extent on his CamelForth. Brad also recommends benchmarking `EXECUTE, OVER, ROT, 0=, +!` and `DODOES`.

The Inner Interpreter

NEXT ENTER EXECUTE EXIT

NEXT
: hands control from one machine code fragment to the next. Because it is used so often, it is vital to optimise it for speed, so it is usually compiled in-line, rather than as a jump to a central routine.

ENTER

the machine code executed by the xt of a secondary definition (compiled by **: "colon"** et al.) ENTER, like NEXT, is a code fragment, not a primitive

EXECUTE
: Forth primitive. Executes the xt on the parameter stack, which can belong to any type of Forth word. Because EXECUTE finishes with a JUMP, it does not need NEXT, which is supplied by whatever it executes.

EXIT
: Forth primitive. Returns from a secondary definition. It is compiled by **; "semicolon"**

Indirect Code Version

Assuming the data field directly follows the code field, DATA is `W CELL +`

NEXT `[IP] TO W CELL +TO IP` (note the order!!) `[W] JUMP`

EXECUTE `POP PSP TO W   [W] JUMP`

ENTER `IP PUSH RSP  DATA TO IP NEXT`

EXIT `POP RSP TO IP  NEXT`

Note how both NEXT and EXECUTE need to update W so the next word to execute can find the right data field address if it needs to.

Direct Threaded Versions

There are two different flavours of direct threaded code: in one the code field of a secondary definition contains `JUMP ENTER` and in the other it contains `CALL ENTER`.

JUMP version

As with indirect code, W holds the code field address. DATA now has to increment past the width of the opcode JUMP xxxx to find the data field address. The only other real difference to the indirect threaded interpreter is that NEXT and EXECUTE now jump directly to W, not to the address pointed to by it.

NEXT `[IP] TO W   CELL +TO IP  W JUMP`

EXECUTE `POP PSP TO W  W JUMP`

ENTER `IP PUSH RSP   DATA TO IP  NEXT`

EXIT `POP RSP TO IP  NEXT`

CALL Version

With the direct threaded version using CALL ENTER , DATA can pop the required address from the hardware stack (where it has be pushed by the CALL), and does not need to reference W at all. Therefore, NEXT and EXECUTE do not need to update it either. Whichever method is chosen, it is vital to be consistent. DATA should work for any word that needs its data field address, whether that word has been invoked by NEXT or EXECUTE. So if DATA depends on the action of CALL, then *every* class of word that requires access to its own data should have a code field of the form CALL xxxx. ENTER and EXIT are unchanged, except of course that they use the new NEXT.

NEXT `[IP] CELL +TO IP JUMP`

(this may be a single opcode on some chips)

EXECUTE `POP PSP JUMP` (likewise)

Native Code Version

Strictly speaking, native code Forths do not have an inner interpreter; it is all machine code, and NEXT is superfluous to requirements. However, a distinction may be made between "primitives" (compiled in-line) and "secondaries" for which a CALL is compiled. Seen this way, IP is the hardware program counter, and RSP the hardware call stack, and the "inner interpreter" looks like this:

ENTER Hardware CALL.

(Compiled in-line in the calling definition - there is no code field as such in a colon definition)

EXIT Hardware Return from Subroutine

NEXT Not needed except to end a "secondary", where it is identical to EXIT

Although colon definitions have no code and data field, the terms do make sense for other "secondary" words, so EXECUTE and DATA are the same as for the CALL version of direct threaded code.

Run-Time Codes for Defining Words

DOCON DOVAR DOUSER DODOES>

Like ENTER, these are not Forth primitives, but the machine code that defines the actions of different types of Forth word.

DOCON `[DATA] PUSH PSP NEXT`

DOCON is compiled by CONSTANT - the data field holds constant's value. **NB** Some Forths may compile constants as literals. The values of these constants cannot be changed once they are defined.

DOVAR `DATA PUSH PSP NEXT`

DOVAR is compiled by CREATE and VARIABLE. Again, some Forths may compile the address of a variable as a literal.

DOUSER `[DATA] UP + PUSH PSP NEXT`

DOUSER is compiled by USER Variables of which every task in a multi-tasking Forth has its own copies, in an area ofmemory pointed to by UP. The data field of a USER holds its offset from the start of that area.

DODOES>
```
IP PUSH RSP
   POP hardware call stack TO IP
   DATA PUSH PSP NEXT
```

DODOES> is perhaps the most subtle piece of code in Forth, associating a data field with an already defined high-level action. This is achieved by prefixing the body of the high-level definition with the native code fragment CALL DODOES> instead of the normal code field for a colon definition. The 'child' word (to which the data field belongs) invokes that instance of CALL DODOES just as it would any other machine code fragment.

Thus, if CALL DODOES is at address xxxx, the code field of the child word contains xxxx if it is indirect threaded, CALL xxxx if it is native code, and either CALL xxxx or JUMP xxxx if it is direct threaded.

The first part of DODOES> is analogous to ENTER, with the address of the thread popped from the hardware call stack, where it had been pushed by the call to DODOES>. The high-level code expects the data field address to be on the parameter stack, so that is pushed before NEXT allows execution to start. Since the address pushed by the extra call has been removed by that stage, the normal DATA finds the correct address for the child's data field. Note that the pseudocode assumes that the hardware stack pointer is *not* one of the virtual registers. If it is, the code will need to be modified. With native code, where the RSP is the hardware call stack, CALL DODOES can simply be replaced by the sequence DATA PUSH PSP.

Literals and Run-Time Words for Control Structures

LIT BRANCH ?BRANCH (DO) (?DO) (+LOOP) (LOOP) (LEAVE)

The compiler for classic threaded code is absurdly simple. All it does is add the xt of each word it finds to the current definition. Any more complex action is dealt with by the fact that some words it encounters are not compiled in this way, but executed instead. For example, `;` compiles EXIT and then turns off the compiler. Such words are called **immediate** words. The following **run-time** words are compiled by the immediate words which build literals and control structures. The immediate words lay down the structure - the run-time words define what to do with it.

Common code fragments

branch `[IP] TO IP` (jump to in-line address)

(or `[IP] +TO IP` if in-line address is relative)

skip `CELL +TO IP` (skip in-line address)

NB Native code may in-line runtime words. In such cases IP will *not* point directly to the in-line value, and adjustment has to be made for the size of the code itself

LIT `[IP] PUSH PSP skip NEXT`

BRANCH `branch NEXT`

?BRANCH `POP PSP  0= IF skip ELSE branch THEN NEXT`

LIT is compiled by literals, to push the in-line value that follows it onto the stack. BRANCH is compiled by the control structure words AGAIN, ELSE and REPEAT - ?BRANCH by IF and UNTIL.

Counted Loops

These have got a little bit baroque in Standard Forth, having to support the following features:

- DO...LOOP can count all the numbers from zero up to the unsigned maximum possible in one cell, therefore 0 0 DO ... iterates max+1 times, and ?DO is needed instead of DO for a loop that iterates zero times.
- DO or ?DO work with both LOOP and +LOOP, as do I and J
- LEAVE exits the loop immediately after the next LOOP or +LOOP
- +LOOP can take either positive or negative increments. In fact, it can vary between positive and negative in the one loop.
- LEAVE exits the loop immediately after the next LOOP or +LOOP

All of this means that testing for termination is tricky. The test is for crossing the boundary between the limit and the limit-1 *from either direction*, so negative-going +LOOPs execute one more time than you might expect. The test is made by offsetting the loop counter by HALFMAX (cell with only the high bit set) - limit, so that overflow is signalled (the sign of the high bit changes) when it terminates.

The behaviour of ?DO and LEAVE has led to some strange implementations. For example, F83 DO pushes the exit address of the loop on the return stack so that it can be picked up and used by LEAVE, and it does so whether that particular loop has a LEAVE or not. This version is more rational. LEAVE compiles its own exit address and doesn't mess with the return stack.

Pseudocode Assumptions

The return stack grows downwards, so [RSP] is the top item, [RSP 1 CELLS +] the next, and so on. It does not matter which way it grows - use whichever is more convenient. You will soon see that it is useful for RSP and PSP to have indexing capability!.

Common Code Fragments

```
setloop ( - start value in W, limit value on stack)
     HALFMAX POP PSP -  PUSH RSP   (amount of offset)
    W [RSP] + PUSH RSP   (adjusted start value)

unloop ( - remove loop parameters)  2 CELLS +TO RSP
```

The Runtime Words

```
(DO)      PSP POP TO W setloop NEXT

(?DO)   POP PSP TO W
   [PSP] W = IF    (does start=limit?)
    POP PSP TO W branch (do not enter loop)
  ELSE setloop skip THEN
  NEXT

(LOOP)    1 +TO [RSP] overflow? IF
       (end of count)
  unloop skip ELSE branch THEN
  NEXT

(+LOOP)  POP PSP +TO [RSP] overflow? IF (end of count)
  unloop skip ELSE branch THEN
  NEXT

(LEAVE)   unloop branch NEXT
```

The corresponding immediate words ?DO, LOOP, +LOOP and LEAVE all compile a branch address. ?DO and LEAVE branch past the end of the loop; LOOP and +LOOP branch back to the start.