PART 2. THE FORTH KERNEL

CHAPTER 4. INTERFACE TO THE HOST COMPUTER

Source code discussed in this chapter is in the file KERNEL86.BLK, screens 3 to 15.

4.1. VIRTUAL FORTH COMPUTER

The Virtual Forth Computer is a program loaded into the memory of a real computer. It partitions the computer memory into areas of specific functionality and enables the real computer to process Forth command streams. Fig. 4.1 is a schematic representation of the functional parts in a Virtual Forth Computer. It consists of a dictionary, two stacks, a terminal input buffer, and a number of disk buffers. These are the essential parts in a Virtual Forth Computer.

The Virtual Forth Computer uses a set of registers to keep the most vital information and to control the execution sequences. They are:

SP	Data Stack Pointer
RP	Return Stack Pointer
IP	Interpretive Pointer
W	Current Word Pointer
PC	Program Counter

The program counter PC and the return stack pointer RP are usually registers in the host CPU. The data stack pointer SP, the interpretive pointer IP, and the current word pointer W can reside in CPU as registers or implemented in memory if the host CPU does not have enough registers.

The dictionary is a linked list of word definitions. Each word definition consists of five fields: The name field and the link field allow definitions to be linked into a linear list which can be searched by the text interpreter for a command by its name. The code field contains the address of the inner interpreter for this definition and the parameter field contains necessary information specific for the task defined for this definition. The view field contains information on where the source code of the definition is located on disk to help user in locating the source code and detailed documentation on the definition.

Two stacks are needed by the Forth virtual machine. The return stack contains a list of addresses of words which are waiting to be executed, or addresses to be returned to after procedure calls. It is similar to the return stacks used in most modern computers. The other stack is called data stack, holding a list of numeric and logic parameters to be passed between words. Separating numeric parameters and return addresses into two stacks allows procedure calls without passing parameters through explicitly parameter lists. It greatly simplifies the syntax of Forth and cuts down the overhead for procedure calls.

Buffers are used to reduce time and efforts to process data transferred between the Forth computer and the I/O devices. Since the two major devices in a typical computer system are the disk for mass storage and the terminal for operator control, two buffer areas, a disk buffer area and a terminal input buffer, are allocated to handle the I/O data.

The kernel of the F83 system is the part of the dictionary which contains words defined in the machine code of the host computer, and transforms the host computer into a Forth computer so that the computer can accept and act upon Forth commands given to it either through a keyboard or through text loaded from a disk. It is the elementary Forth operating system which can be expanded by loading utility programs and application programs and executing those programs.

4.2. FORTH COMPUTER HOSTED ON 8086

The Virtual Forth Computer is a hypothetical computer to illustrate the architecture or the ideal structure of a computer which can be used to implement the Forth instruction set. So far, we have yet to see a computer built based upon this architecture. However, this architecture can be implemented on any CPU worthy of the name. As a matter of fact, most of the commercial CPU's have at least one version of Forth on them, including all popular microprocessors, minicomputers, and many mainframes. Because Forth is simple and relatively small, it can be implemented on a computer with about one man-month's effort. This is very short comparing to other operating systems or high level languages.

F83 has three versions: one for 8080, one for 8086/8088, and one for 68000. Because of the very small number of registers that are available in the 8080 CPU, many Forth registers will have to be simulated in memory. Most numbers processed in Forth are 16 or 32 bits in width. They have to be manipulated in 8 bit chunks in 8080. Forth code definitions in 8080 machine codes are thus very messy and very difficult to explain. Between 8086 and 68000, my personal preference is 68000 which has a much cleaner architecture and more orthogonal instruction set. Nevertheless, I feel very feeble in raising a voice against the infinite wisdom of IBM, who picked 8088 for PC. Since there are apparently more people using 8088 than 68000, it is better to write this manual in terms of the 8086/8088 F83 model if I want to sell more copies of this manual. .new .56 Fig. 4.1. The virtual Forth computer .new ASSIGNMENTS OF FORTH REGISTERS IN 8086

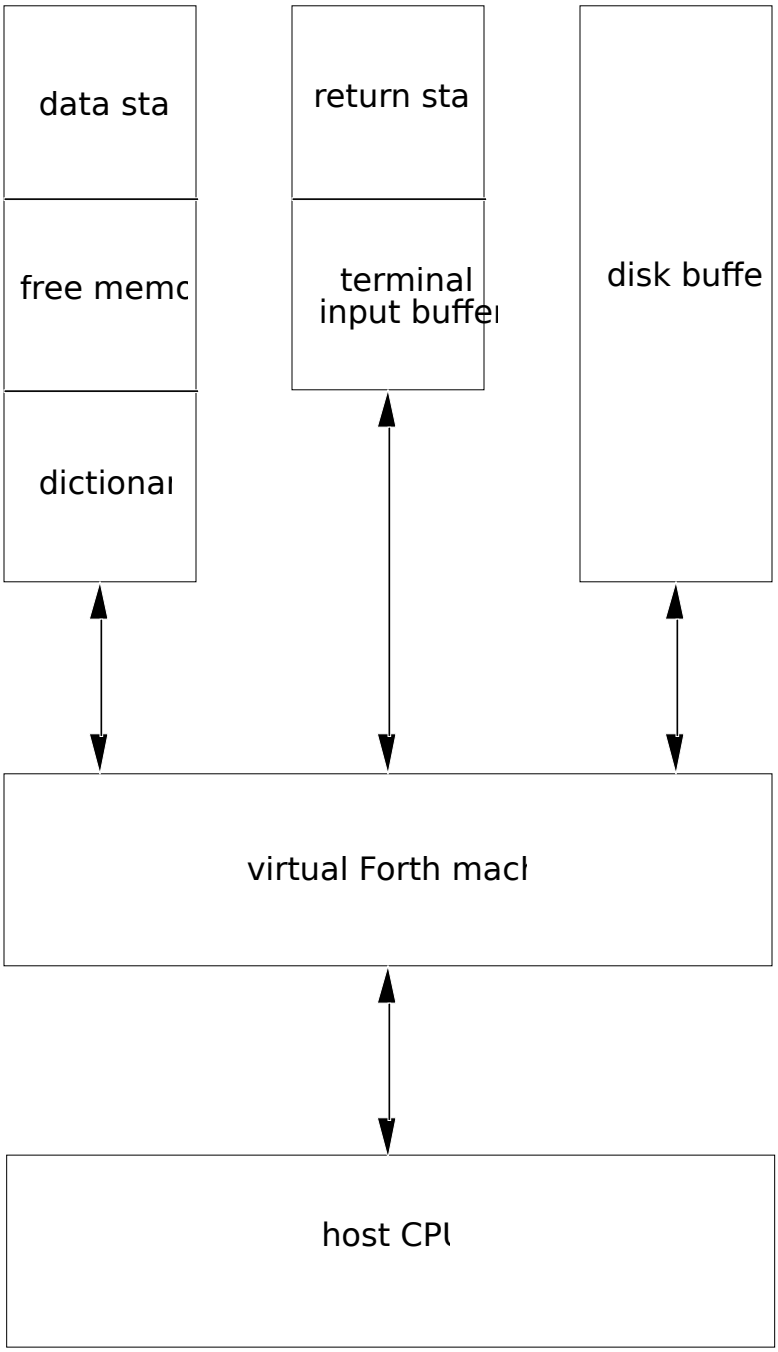
First of all, let us see how the Forth registers are assigned in the 8086 CPU:

TABLE 4.1. 8086 REGISTER ASSIGNMENTS FOR FORTH

8086 Reg.	FORTH Reg.	Function
AX		Accumulator
CX		Scratch, counter
DX		Scratch, I/O control
BX	W	Current word pointer
SP	SP	Data stack pointer
BP	RP	Return stack pointer
SI	IP	Instruction pointer

DI	Scratch
ES	Extra segment
CS	Code segment
SS	Stack segment
DS	Data segment

Figure 4.1 The virtual Forth computer



8086 has only one stack, the data stack, which allows pushes and pops. Other registers do not have automatic incrementing or decrementing facilities, and increment/decrementing must be done explicitly. An interesting exception is the SI index register. When SI is used in the supposed string instruction LODS and STOS, it is incremented by 1 or 2 bytes to point to the next string element. It is ideal for the interpretive pointer. The stack pointer SP is used to implement the Forth data stack, while the Forth return stack is simulated using the BP register. Popping and pushing on the return stack have to be done explicitly by incrementing and decrementing the BP register. The word pointer W is simulated by the BX register. Lacking automatic incrementing and decrementing facility, the W register has to be left pointing to the code field after NEXT. It has to be incremented in code so that it will point to the parameter field. We will have to use indirect JMP instructions through the code field to control the program flow.

Other 8086 registers, like AX, CX, DX, and DI, can be used freely in code routines. However, parameters and other information cannot be passed from one definition to another through these registers. They have to be initialized appropriately before use, but they do not have to be restored before the end of a code definition. The Forth W register (the 8086 BX register) contains the code field address of the definition under execution. If this address is not needed in the code definition, this register can be also used without restoring. The SP, RP, and IP registers, however, have to be restored to the original values if they have to be used in a code definition.

All information in the F83 system is contained within a single 64K byte memory segment, and the four segment registers ES, CS, SS, and DS are initialized to the same segment. They can be changed to address other segments of memory, but must be restored before the end of a code definition. The F83 system does not use them.

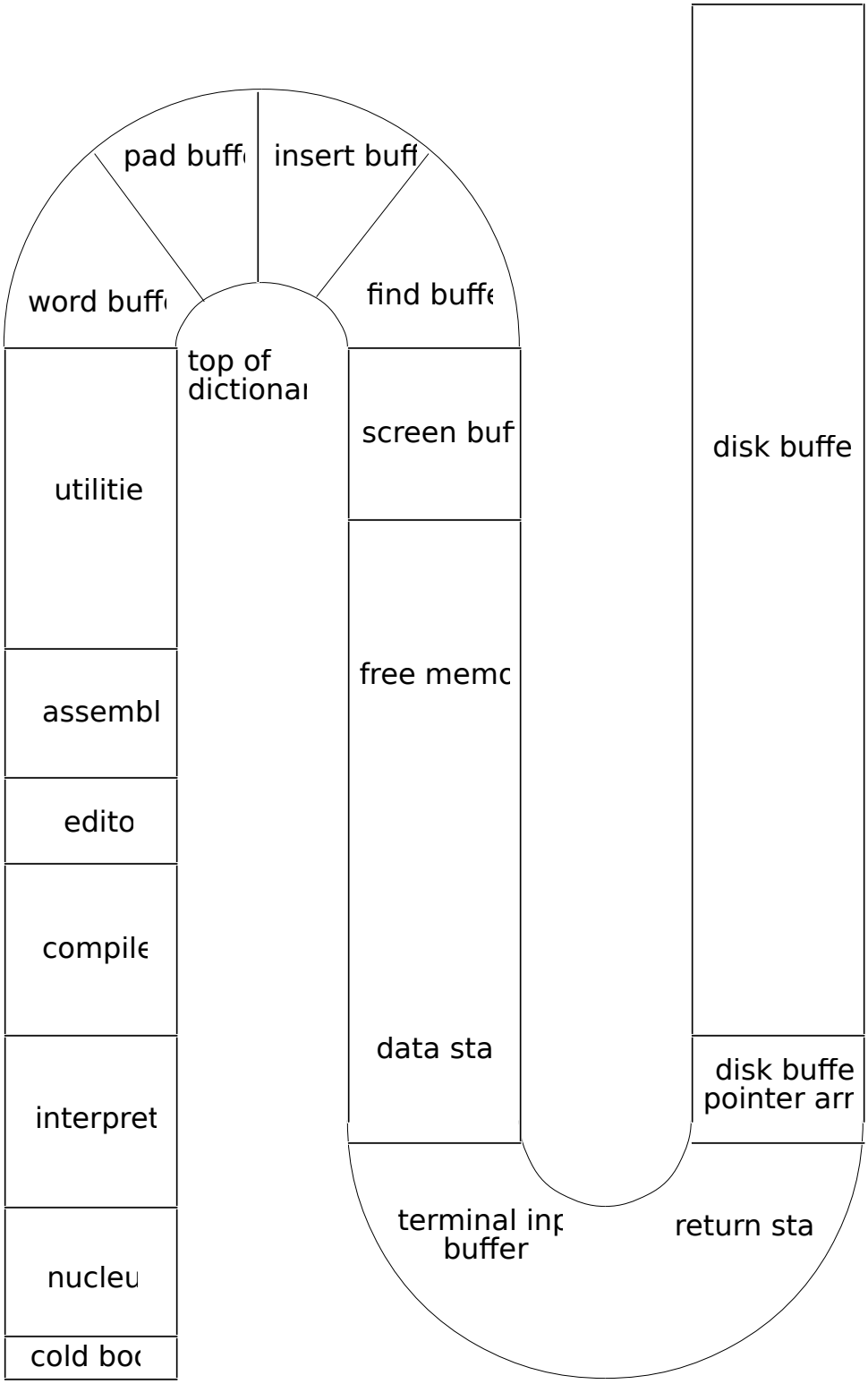
MEMORY MAP

RAM memory in the host computer, as used by the F83 system, is a contiguous 64 Kbytes of memory. Forth separates this memory space into a few regions, each dedicated to specific function. The lowest memory is used to hold the interrupt vectors which are used by the hardware to service external interrupts and software interrupts. Immediately above the vector region is the dictionary, holding all the executable codes of the word definitions. Above the dictionary is a free space for the user to define new words. On the top of the memory map is the region for disk buffers. F83 allocate 4 Kbytes for the buffers, enough to hold 4 blocks of data from/to the disk. Under the disk buffers is an area storing user variables which are essential parameters for the Forth system to work. Below the user area is the return stack, sharing its space with the terminal input buffer, which is used to store characters received from the terminal keyboard before processed by the text interpreter.

Below the terminal input buffer is the data stack, growing downward into the free memory space above the dictionary. The space just above the dictionary are used to store temporary text data. We can identify a word buffer, a text buffer called PAD, an insert buffer and a delete buffer. The later two buffers are used by the text editor. A video buffer of 1 Kbytes is also assigned if the screen editor is invoked. These buffers float on the top of the dictionary, moving to higher memory as new words are added to the dictionary. Data stored in them have to be used before new definitions are defined.

Fig. 4.2. schematically shows the arrangement of various regions in a typical F83 Forth system. Most Forth system are arranged similarly.

Figure 4.2 Memory map of F83 system



4.3. INNER INTERPRETERS

Inner interpreters in Forth are a set of execution procedures, usually in the machine code of the host computer, which execute various Forth words by processing the information stored in their parameter fields. The address of such a procedure is stored in the code field of a word definition. Forth definitions of the same class have the same address in their code fields. The inner interpreters are also called 'runtime routines'. Two major inner interpreters are used to process code definitions, defined by machine instructions, and colon definitions, defined in terms of other existing Forth words. Many other minor inner interpreters are used in F83 system to process constants, variables, user variables, and other types of data and structures.

CODE INTERPRETERS

Forth words defined by host machine instructions are executed by one of two routines, EXECUTE or NEXT. EXECUTE can be used to start executing any Forth word, given that the code field address of the Forth word is placed on the data stack before calling EXECUTE. EXECUTE is a regular Forth word which can be executed interactively or called from the text interpreter. At the end of all code definitions, there must be a jump to the machine routine labelled NEXT, which transfer control to execute the next word in the execution sequence. In F83, NEXT is centralized so that every word must return to it through a JMP instruction. NEXT assumes that the code field address of the next word to be executed is stored in the IP register.

```
CODE EXECUTE    ( cfa --- )     Execute the word whose execution address is on the data
                                stack.
        W POP                   Pop execution address into W.
        0 [W] JMP               Make an indirect jump through W.  W is left pointing to the END-
CODE                            code field.  It must be incremented if parameter field must be
                                addressed.
LABEL DPUSH                     A label in the target system.
        DX PUSH                 Push contents of DX on the data stack.
LABEL APUSH                     Another label.
        AX PUSH                 Push contents of AX on the data stack.
LABEL >NEXT                     The principal point of return for all code definitions.  The
                                execution address of the next word to be executed in in IP
                        register.
        AX LODS                 Load the next execution address from IP into AX and
                                increment IP.
        AX W MOV                Copy execution address into W.
        0 [W] JMP               Indirect jump through W.
```

The codes of EXECUTE appears in Screen 36 of META86.BLK, and the codes of NEXT is in Screen 25 in the F83 source. The fact that these codes are scattered in many separated blocks makes it difficult for the reader to put a whole picture together. It is the purpose of this manual to present to the reader a well organized system description to help him understand the F83 model fully.

```
: NEXT                          A macro definition to assemble a jump to >NEXT instruction
                                at the end of a code definition.
        >NEXT #) JMP            Assembler the jump instruction.
```

```
        ;
: 2PUSH                         Another macro definition.
        DPUSH #) JMP            Jump to DPUSH, push DX on stack.
        1PUSH                   Macro definition pushing AX on the data stack before
        NEXT.
        APUSH #) JMP
        ;
```

These definitions are defined in the metacompiler as shown in the source code. Let us not worry about them here.

ADDRESS INTERPRETER

The address interpreter is used to execute a high level Forth definition whose parameter field contains a list of execution address. It processes this list by executing words at these addresses sequentially.

The address interpreter is not an executable Forth word. It is a machine code routine labelled NEST:

```
LABEL NEST              IP has the address to return and W has the code field address
                        of the colon definition to be called.
        W INC   W INC           Increment W to point to the parameter field of the callee.
        RP DEC   RP DEC         Decrement return stack pointer and prepare for a push.
        IP 0 [IP] MOV   Push contents of IP, the return address on the return stack.
        W IP MOV                Copy the first execution address into IP, to start the called
                        colon definition.
        NEXT            Assembler >NEXT #) JMP here.
```

In 8086, W and RP have to be incremented twice because these registers are byte pointers. All code routines end up with the code >NEXT #) JMP. This is very convenient in debugging the system or changing the behavior of the code interpreter to include new features in the Forth system.

```
CODE EXIT                       Terminate a colon definition and return to the caller routine
                        whose address is on top of the return stack.
        0 [RP] IP MOV           Pop the return address back to the IP register.
        RP INC  RP INC
        NEXT            Return.

CODE UNNEST   ' EXIT   ' EXECUTE !
```

The standard word EXIT is vectored to UNNEST which is the reverse of NEST. NEST is equivalent to the high level SUBROUTINE call in FORTRAN, and UNNEST is the equivalent of RETURN.

VARIABLE INTERPRETER

The variable interpreter uses the W register to point to the parameter field of the variable definition and returns the parameter field address on the data stack for the subsequent words to access the parameter field. This inner interpreter can be shared by other types of definitions which use the parameter fields to store various types of data, like strings, double integers, floating point numbers, or even large arrays.

When a new definition is created in the dictionary, the compiler assumes that the definition is of this type unless a new inner interpreter is defined. Instead of the old name DOVAR in fig-Forth, F83 uses the generic name DOCREATE:

```
LABEL DOCREATE          W points to the code field.
      W INC   W INC     Increment W to point to the parameter field.
      W PUSH            Push pfa on the data stack.
      NEXT        Return.
```

W register as returned by NEXT contains the code field address of the variable definition. To get the parameter field address, W has to be incremented here. This is called the post-incrementing NEXT. In many other Forth systems, the W register is incremented inside NEXT so that it points to the parameter field at the end of NEXT. The pre-incrementing NEXT is more desirable than the post-incrementing NEXT, because the post-incrementing NEXT requires that the code field is two bytes ahead of the parameter field. Because F83 uses the W register to make the indirect jump to the inner interpreter, the W register cannot be incremented before the jump.

CONSTANT INTERPRETER

The constant interpreter is very similar to the variable interpreter. The only difference is that the constant interpreter returns the contents of the parameter field while the variable interpreter returns the address of the parameter field.

```
LABEL DOCONSTANT        W points to the code field.
      W INC    W INC    Point W to the parameter field.
      0 [W] AX MOV      Fetch contents in the parameter field to AX register.
      1PUSH JMP         Push AX on the data stack and then return.
```

The constant interpreter first copies the contents of the parameter field into the AX register, and then jumps to the APUSH routine which pushes AX on the data stack before falling into NEXT.

USER VARIABLE INTERPRETER

User variables are defined in order to make a multitasking or multiuser Forth system such as F83. These variables are not addressed by their parameter field addresses, but by an offset into a memory area unique to the current user, a user variable area whose starting address is stored in a register or a variable UP. The user variables define the operating environment for a user at any point of its operation. Since each user has its own user variables preserved in a unique memory area, users or tasks can be switched very conveniently with mininmal house keeping.

The user variable interpreter in F83 is defined as:

```
VARIABLE UP             The user area pointer is defined as a variable.

LABEL DOUSER-VARIABLE
      W INC   W INC     Point W to the parameter field.
      0 [W] AX MOV      Get the user area offset from the parameter field.
      UP #) AX ADD      Add the offset to the base address in UP.
```

```
1PUSH           Push the address of the user variable on the data stack and
                return.
```

The parameter field of a user variable stores the offset value of the user variable in the user area. This offset value is added to the starting address of the user area as stored in the variable UP. The address returned on the data stack is the address of the user variable of the current user who is controlling the Forth system at this moment.

With all the viable system parameters saved in the user variable areas, the task switching in a Forth multitasking system is very easy and very efficient. The multitasker only has to save and restore the IP, RP, and SP in between two tasks. We will get into this in detail later. F83 has a very interesting multitasker which is a good demonstration of the power and the versatility of Forth as a system and as a language.

HIGH LEVEL INNER INTERPRETER

Inner interpreters are preferably coded in the host machine code, because they are the actual routines executed by the host computer. However, Forth does provide the CREATE ... DOES> construct for users to define inner interpreters using high level Forth words. These high level inner interpreters are easy to develop and eminently transportable across different host computers. The mechanism which allows this type of inner interpreters to execute correctly is DODOES:

```
LABEL DODOES        W points to the code field of the current words and SP
                    points to the high level inner interpreter.
    SP RP XCHG
    IP PUSH         Push current IP on the return stack.
    SP RP XCHG
    IP POP          Pop address of the high level interpreter into IP.
    W INC   W INC   Point W register to parameter field which may contain data.
    W PUSH          Push W on the data stack.
    NEXT            Returnto execute the high level interpreter while the top item
                    on the data stack points to the parameter field of the current
                    word.
```

Using this DODOES, the new words defined by the CREATE...DOES> structure are almost identical to those defined by the CREATE...;CODE structure. DODOES must be the first word to be executed in the high level inner interpreter.

DEFERRED WORD INTERPRETER

F83 uses a special technique to handle forward references, which is normally not allowed in a regular Forth system. A deferred word is created with a blank parameter field. When the contents of the deferred word is finally compiled, the parameter field in the deferred word is then patched with a pointer pointing to the beginning of the compiled codes so that the deferred word can be executed. Before the contents of a deferred word are defined, however, the deferred word can be referred to by the compiler and be compiled as other regular words even if it cannot be executed. This technique is useful, especially during metacompilation when words have to be referred before their functionality can be precisely defined by the words metacompiled after them.

The deferred word interpreter fetches the address in the parameter field and makes an indirect jump through it:

```
LABEL DODEFER            Execute the word whose execution address is stored in the
                         parameter field of this deferred word.
    W INC   W INC        Get the parameter field address.
    0 [W] W MOV          Replace W with contents of the parameter field.
    0 [W] JMP            Make an indirect jump through it.
```

The deferred address can also be stored as a user variable so that each user may have its own version of the execution procedure to be referred to by the same name.

4.4. INTERPRETERS FOR IN-LINE DATA AND STRINGS

In the parameter field of a colon definition there is normally a list of execution addresses, which is scanned sequentially by the address interpreter and executed. However, there are many instances that the execution sequence must be changed in runtime or that some special data have to be included in-line with the execution addresses, like literal numbers and character strings. A set of special words is defined to take care of these conditions at runtime, when the colon definition is being executed. Although these special words are given names like other definitions and can be found by both the text interpreter and the colon compiler, they are not meant to be invoked by either. They are compiled into colon definitions by a corresponding set of immediate words or compiler directives. To indicate their associations with corresponding compiler directive and that they are not to be directly invoked, they are assigned names with enclosing parentheses. Executing them interactively from a terminal is the most convenient way to crash a Forth system. Be warned of it!

```
CODE (LIT)   ( --- n )   Push the contents in next cell on the data stack.
    AX LODS              Load the contents of next cell, pointed to by IP, into AX.
                         Increment IP to skip over the numeric literal.
    1PUSH                Push the literal number on the data stack and return.
```

LODS is an interesting 8086 instruction. It is used to access character strings in memory using the SI register as a pointer. After the memory fetching, SI is automatically incremented. It happens that the SI register is the IP register in Forth virtual computer and the incrementing is exactly what we wanted in (LIT). It makes an extremely simple code definition for (LIT). APUSH pushes the contents of AX register on the data stack before falling into the NEXT routine.

(LIT) thus overrides the natural tendency of the address interpreter to interpret data as execution addresses and forces the interpretation of the contents in the next cell as an in-line literal. This is the way numbers are compiled in a colon definition, preceded by (LIT), so that in runtime, the number will be pushed on the stack and not to be mistaken for an execution address.

```
: (.")   ( --- )         Print the next character string to the terminal.
    R>                   RP is pointing to the next cell where the string starts. Pop
                         the string address to data stack.
    COUNT                Get the string address and character count on the stack.
    2DUP + EVEN          Compute the address of the next executable word after the
```

```
                    string.
    >R                      Replace the next execution address back on the return stack.
    TYPE  ;                 Now, type out the string.
```

(.") and the character string following it are compiled by the immediate word ." , in-line with the other execution addresses in a colon definition. When the colon definition is executed, (.") will pull this string out of the execution sequence, print it on the terminal, and then pass the control to the word after the string. This is the way we let a colon definition print messages on the terminal to facilitate the user-computer interface. Computers can be make much more friendly this way if proper messages are printed timely.

```
: (")          ( --- addr n )       Leave the address and the character count of the following
                                    string on the stack and continue execution after the string.
    R>  COUNT                       Get the address and count on stack.
    2DUP  +  EVEN                   Compute the next executable word address,
    >R  ;                           and put it back on the return stack.
```

(") is very similar to (.") in the way it handles the in-line string and the execution sequence. The difference between them is that (") leaves the string address and character count on the data stack without doing any terminal output; therefore, the string data can be manipulated any way we want in the colon definition.

4.5. INTERPRETERS FOR CONTROL STRUCTURES

BRANCH AND ?BRANCH

We all think Forth is a totally structured programming language, even saying: "Look Mom, no GOTO's!" GOTO's are replaced by structures like IF ... ELSE ... THEN , BEGIN ... UNTIL , and DO ... LOOP , etc. Well, the hard truth is that Forth does have GOTO's, disguised in names like BRANCH and ?BRANCH, and many other words. If you learned how to use them, you could jump anywhere you wanted and create really messy spaghetti codes. Novices are made to believe Forth is GOTOless because they are shielded from the dark side of Forth.

BRANCH and ?BRANCH take the contents in the next cell as the address of the next executable word and direct the address interpreter to that address to start a new execution sequence. This can be done simply by manipulating the interpretive register IP.

```
CODE BRANCH     ( --- )         Perform an unconditional jump to the address in the next
                                cell.
LABEL BRAN1
    0 [IP] IP MOV               Copy next cell into IP, thus
    NEXT                        effecting the branch.
    END-CODE

CODE ?BRANCH    ( f --- ) If the flag on stack is false, branch to the next address;
                          otherwise, skip the next cell and continue the execution
                    sequence.
```

```
        AX POP              Pop the flag into AX register.
        AX AX OR            Set the CPU status register.
        BRAN1 JE            Branch if flag is false.
        IP INC   IP INC     Skip the jump address if flag is true.
        NEXT
        END-CODE
```

BRANCH is compiled by ELSE, REPEAT, and AGAIN. ?BRANCH is compiled by IF, WHILE, and UNTIL. The cell immediately following BRANCH or ?BRANCH is the address of the next executable word in memory, and it directs the conditional or unconditional branching, deviating from the normal sequential execution path favored by the address interpreter.

THE NEW F83 LOOPS

The DO-LOOP structure experienced a major surgery in the birth of Forth-83 Standard, drastically deviated from the DO-LOOP structure that Charles Moore invented. The basic reasons behind the new DO-LOOP structure were to eliminate the discontinuity of indexing through the 8000H boundary and to leave the loop immediately at LEAVE. F83 provides a solution by using three numbers on the return stack to handle the indexing and looping. The number at the bottom of the three is the address of the word right after LOOP, providing LEAVE with the return address to terminate the looping. The second number is the loop limit, offset by 8000H so that the index range from 0 to FFFFH becomes contiguous. The top number is the difference between the index and the limit, also offset by 8000H. At the end of the loop, LOOP increments the top number on the return stack by either one or the amount specified in the case of +LOOP, and tests for overflow from bit 14 to bit 15. The overflow condition occurs when the 8000H boundary is crossed from either direction. Therefore, both the positive and negative increments are handled correctly with a single run-time loop routine. Since the address of the word after LOOP is carried on the return stack, LEAVE can use this address to jump out of the loop.

```
CODE  (DO)    ( limit index --- )     Push the in-line exit address and the modified loop limit
                                      and scan range on the return stack.
        AX POP                        Get the index.
        BX POP                        Get the limit.
LABEL PDO
        RP DEC   RP DEC               Make room on the return stack.
        0 [IP] DX MOV                 Get the in-line address following DO.
        DX 0 [RP] MOV]                Push the exit address on the return stack.
        IP INC   IP INC               Pointing IP to the next executable word.
        8000 # BX ADD                 Offset the limit by 8000H.
        RP DEC   RP DEC               Make more room.
        BX 0 [RP] MOV]                Push the modified limit on the return stack.
        BX AX SUB                     Subtract limit from index, also offset by 8000H.
        RP DEC   RP DEC               Make room.
        AX 0 [RP] MOV                 Push the index scan range on the return stack.
        NEXT                  All done.
        END-CODE

CODE (?DO)    ( lim ind ---)          Same as (DO) except that if index is the same as limit, the
                                      entire loop is skipped.
```

```
        AX POP              Index.
        BX POP              Limit.
        AX AX CMP           Compare index and limit.
        PDO JNE             If not equal, execute the loop.
        0 [IP] IP MOV       If equal, jump over the do loop.
        NEXT END-CODE
```

With the modified index, modified limit, and the exit address on the return stack, the task for end-of-loop routines is much easier. Believe it or not, this new loop structure is claimed to run faster than the old, traditional loop.

```
CODE  (LOOP)      ( --- )     Branch back to the executable word after DO if the index
                              does not cross the 8000H boundary.  If it does, exit the loop
                              after clearing the return stack.
        1 # AX MOV            Increment by one. LABEL PLOOP Increment top of return
                              stack,
        AX 0 [RP] ADD         the scanning index.
        BRAN1 JNO             If overflow condition is not set, jump to the in-line address
                              compiled after (LOOP) and repeat the loop.
        6 # RP ADD            Pop all three numbers off the return stack.  Clean up the
                              return stack to the state before the do-loop.
        IP INC   IP INC       Point IP to the next executable word.  Exit the loop.
        NEXT   END-CODE

CODE  (+LOOP)     ( inc ---)  Increment the scanning index by the value on the data stack
                              and decide whether or not to loop.
        AX POP                Get the increment.
        PLOOP #) JMP          Use the same loop routine in (LOOP).
        END-CODE
```

Since the scanning index on top of the return stack is not the index as we understood, the functions of I and J are also different.

```
CODE  I       ( --- index )   Return the current loop index.
        0 [RP] AX MOV         Get the scanning index on top of the return stack.
        2 [RP] AX ADD         Add the modified limit to the scanning index.  The result is
                              the actual current index.
        1PUSH                 Push it on data stack.
        END-CODE

CODE  J       ( --- index )   Return the loop index of the next outer loop in nested do-
                              loops.
        6 [RP] AX MOV         Get the outer index.
        8 [RP] AX ADD         Add the outer limit.
        1PUSH                 Push the computed index on stack.
        END-CODE
```

THE NEW LEAVE

The Forth-83 Standard requires that when LEAVE is executed inside a loop, the loop be exited immediately. It was agreed that the old LEAVE is not desirable in allowing execution to continue to the next LOOP before exiting the loop. Unwelcome guests should not be permitted to remain when the party is over. Since the exit address of the word after LOOP is compiled after (DO) and tucked on the return stack, LEAVE can be executed using this piece of information:

```
CODE  (LEAVE)       ( --- )          Immediately exit a DO-LOOP.
LABEL PLEAVE
        4 # RP ADD                   Pop the index and limit off the return stack.
        0 [RP] IP MOV                Copy the exit address to IP, ready to exit the loop.
        RP INC   RP INC              Clear the return stack.
        NEXT   END-CODE

CODE  (?LEAVE)      ( f --- ) Exit the loop immediately if the flag on stack is true.  If not,
                              continue the looping.
        AX POP                       Get the flag.
        AX AX OR                     Test the flag for zero.
        PLEAVE JNE           True.  Leave the loop.
        NEXT                 False.  Continue.
        END-CODE
```

LEAVE is not very useful all by itself because it will defeat the purpose of a do-loop. In most cases, it is used after a testing condition like IF. ?LEAVE combines the functions of IF and LEAVE, and is a much more useful word.