CHAPTER 5. THE FORTH NUCLEUS

The source code discussed here is in the file KERNEL86.BLK, screens 16 to 37.

In the last chapter on the Virtual Forth computer, what we discussed was the 'hardware' of this conceptual computer, such as the registers, the memory and its organization, buffers, and stacks. The inner interpreters are similar to the CPU in this computer, which cause the machine to perform the most primitive operations like jumping from one instruction to the next. There is also a 'software' part of the Virtual Forth computer, i. e., the primitive instruction set or the elementary operations from which programs can be constructed to solve complex, real life programming problems. This primitive instruction set, the counterpart of the microcodes or random logic machine instruction set in a real, conventional computer, is what we mean by the Forth Nucleus. In a real Forth computer, this instruction set will probably be microcoded or committed to random logic in the Forth CPU. Before that becomes a reality, the Forth Nucleus will have to be implemented on a real CPU using its native machine codes.

F83 is available in three versions: one for 8080, one for 8086/8088 and one for the more recent 68000. It' is a pain to discuss the Forth nucleus in 8080 machine code, because we have to pretend that the 8 bit 8080 is a 16 bit machine. There is so much noise in the 8080 codes that you can hardly hear the beautiful music played in Forth. The 8086 is far from being a dream machine. Being a 16 bit machine with more than enough registers in the CPU, the Forth Nucleus put on it looks much nicer and the code is considerably shorter. For most of the instructions in the Forth nucleus, the 8086 code is less than 1 line in length and the functions are fairly obvious. In fact, most code is simple enough that I really don't have to go through it line by line, as I did for the inner interpreters. I will only go through the code by functional groups, making some occasional comments on special features in the F83 implementation.

I encourage you to read the code in the nucleus carefully because it is a good example of assembly programming in Forth. There are lots of techniques and styles we can learn from this code. When you want to write code definitions to take advantage of the speed and to tackle some hardware facilities, the best way is to pick up a code definition in the nucleus of similar functions and modify it to suit your need. Once you are at home with the manipulation of stacks and the CPU registers in Forth assembly style, you will be able to build your own castles.

5.1. 8086 ASSEMBLY LANGUAGE IN FORTH

Assembly code in Forth is quite different from the normal assembly code in a conventional assembler. The most eye catching difference is that the Forth assembly code is written in reverse Polish notation, i.e., operands preceding the operator. The reason is simple. In Forth, the assembler is not a gigantic program which assembles mnemonic codes line by line. The assembly functions are scattered in many small pieces of Forth definitions which are given assembly mnemonic names. When a Forth definition like MOV is executed, it compiles a machine code into the dictionary where we are building the parameter field of a code definition. When MOV is executed, it needs information like source register, destination register, and address mode. This information, the operands, is provided on the data stack prior to the invocation of MOV. MOV takes the operand information from the data stack, does some computation to derive the correct machine code, and compiles this code into the top of the dictionary. All the other assembly definitions do similar things, using data from the stack and compiling specific

codes into the dictionary.

There is a major difference between the Forth colon compiler and the Forth assembler, even though they both build new definitions in the dictionary. When compiling colon definitions, the Forth computer is in the compiling mode, in which words parsed out from the input stream are not executed, but have their addresses added to the dictionary. During assembly, the Forth computer is in the interpretive mode, in which all the assembly definitions are executed. The net result produced by the execution of an assembly definition is that a machine code is added to the dictionary. In other words, we can claim that it is the Forth text interpreter who does the assembly of machine codes. The full Forth system, with all its resources, is supporting the process to assemble machine codes. In a way, the assembly process is so much more complicated than compiling colon definitions that it indeed needs the support of the whole Forth system. The complexity of the assembler is best seen in the actual codes of the Forth 8086 assembler, which will be the subject of Chapter 24. At this moment, we just have to learn how to read the Forth assembly code in the nucleus.

5.2. CODE DEFINITIONS IN THE FORTH NUCLEUS

In the F83 Nucleus, all the code definitions are written in the following general format:

CODE <name> < operands and assembly mnemonics > <end> END-CODE

A code definition is enclosed between two words CODE and END-CODE. Immediately following CODE is the name given to the definition. After the name, there is a sequence of words which are either assembly mnemonics or operands used by the mnemonics. The assembly mnemonics are Forth definitions which assemble machine codes into the parameter field of the code definition under construction. The word before END-CODE is a special word which returns control to the routine which calls the definition in runtime. Anywhere inside or outside of the code definition, comments are placed between (or (S and), which are ignored by the Forth interpreter which does the assembly.

The assembly mnemonics are mostly the same as those mnemonics used in the regular 8086 assembler provided by Intel. However, they are not just names of machine code, they are actually Forth definitions which assemble machine code into the dictionary when they are interpreted or executed. Many of these mnemonic definitions require operands, which are supplied before the mnemonic definitions. If two operands are needed, the format is:

<source operand> <destination operand> <mnemonics>

A partial list of the mnemonic definitions is:

MOV PUSH POP JMP JE JNE JCXZ ADD SUB MUL DIV AND OR XOR
MOVS PUSHF REPZ SAHF WAIT LODS XLAT

The following registers are defined in F83 for 8086:

AL CL DL BL AH CH DH BH
AX CX DX BX SP BP SI DI
ES CS SS DS

Forth registers RP, IP, and W are equivalent to the 8086 registers BP, SI, and BX, respectively.

Several registers are often used for indirect addressing. The indirect addressing operands are the following:

[RP] [IP] [W] [SI] [DI] [BP] [BX]

An offset number must precede the indirect addressing operand. Numeric values needed as operands must be used with a numeric operator following immediately:

#) S#)

where # is preceded by an immediate constant, #) is preceded by an address, and S#) is preceded by an address for intersegment jump.

Three most frequently used code endings are NEXT, 1PUSH, and 2PUSH. They are assembly macros which return control to the next definition in the execution sequence. 1PUSH pushes the AX register on the stack before jumping into NEXT, and 2PUSH pushes first the DX register and then jumps to 1PUSH. Sometimes a JMP is used as a code ending. The routine jumped to must eventually fall into NEXT so that the execution can be continued.

5.3. EXAMPLES OF CODE DEFINITIONS

The following are a few simple examples of the code definitions. They are fully commented here for the purpose of demonstrating the Forth assembly syntax. Since the 8086 has most of the functions required by Forth in machine codes, the code definitions in the F83 nucleus are fairly simple and obvious. I will not try to make dumb comments any more.

```
CODE  @       ( addr --- n )          Fetch a 16 bit value from addr.
        BX   POP                      Pop addr into BX register.
        0 [BX] PUSH           Push the contents of addr, indexed by BX with 0 offset,
                        onto the data stack.
        NEXT                  Jump to next and return.
        END-CODE                      End of code definition.

CODE  !       ( n addr --- )          Store a 16 bit value at addr.
        BX POP                         Pop addr to BX register.
        0 [BX] POP                    Pop n into memory at addr.
        NEXT   END-CODE


CODE  C@     ( addr --- char )        Fetch an 8 bit value from addr.
        BX POP                        Pop addr into BX register.
        AX AX SUB                     Clear the 16 bit AX register.
        0 [BX] AL MOV                 Copy one byte at addr to AL.
        1PUSH                         Push the byte value on stack and return.
        END-CODE

CODE  C!      ( char addr --- )        Store an 8 bit value at addr.
```

```
BX POP
AX POP                  Pop char into AX.
AL 0 [BX] MOV           Store byte into addr.
NEXT   END-CODE
```

Other code definitions in the nucleus are fairly straight- forward and are also adequately commented in the shadow screens. They are grouped together and shown here for reference. I encourage you to read the detailed code and comments in the source listing.

MEMORY COMMANDS

@	!	C@	C!	CMOVE	CMOVE>	
FILL	ERASE	BLANK	MOVE	HERE	PAD	

STACK COMMANDS

SP@	SP!	RP@	RP!	DROP	
DUP	SWAP	OVER	TUCK	NIP	
ROT	-ROT	FLIP	?DUP	R>	
>R	R@	PICK	ROLL		

LOGIC COMMANDS

AND	OR	XOR	NOT	TRUE
FALSE	CSET	CRESET	CTOGGLE	ON
OFF				

ARITHMETIC COMMANDS

+	-	ABS	+!	2*
2/	U2/	8*	1+	2+
1-	2-	UM*	U*D	UM/MOD
*D	M/MOD	MU/MOD	*	/MOD
/	MOD	*/MOD	*/	

COMPARISON COMMANDS

0=	0<	0>	0<>	=
<>	?NEGATE	U<	U>	<
>	MIN	MAX	BETWEEN	WITHIN

DOUBLE INTEGER COMMANDS

2@	2!	2DROP	2DUP	2SWAP
2OVER	3DUP	4DUP	2ROT	D+
DNEGATE	S>D	DABS	D2/	D-
D0=	D=	DU<	D<	D>
DMIN	DMAX			

STRING COMMANDS

COUNT	LENGTH	-TRAILING	UPPER	COMP
CAPS-COMP	COMPARE			