CHAPTER I. THE NOVIX NC4000 CHIP

1. Historical Background

In the beginning, Chuck Moore invented Forth as a programming language to make himself a more productive programmer. It was in the late 1960's when various aspects of Forth were developed and integrated into a single, unified, and complete software development tool and environment. In 1972, Chuck and a few of his colleagues left National Radio Astronomy Observatory (NARO) where they nurture Forth into its present form, and formed Forth, Inc. to explore its commercial applications. For a while, he was content to use Forth as a software tool to solve real world problems, while leaving it to the hardware engineers to pick up the Forth architecture and implement Forth engines.

Forth didn't become a household name in computer circles until Forth Interest Group was formed in the San Francisco Bay area, which distributed fig-Forth source code by the ton at cost, beginning in 1978. Although Forth had established a sizable following in the microcomputer user community, the microcomputer industry was very slow in accepting Forth as an architecture for hardware design and implementation. There were scattered efforts toward building Forth engines using bit slice technology and random logic, but the consequence was almost nil. Meanwhile, Chuck became restive and took it upon himself the task of casting Forth into silicon. In 1980, Chuck left Forth, Inc. to pursuit his new dream.

In the reorganization, Forth, Inc. expanded its board of directors to include Bill Ragsdale, the founding father of Forth Interest Group, and John Peers of Logical Machines, Both of them had very intense interest in seeing Forth put into hardware. The right mix of personality was thus gathered for the precise chemistry necessary for brewing a Forth chip.

At that time, Chuck was designing a prototype board which would execute Forth words as primitive instructions with Glenn Haydon and others. The spark that initiate the Forth chip development was set by Don Colburn of Creative Solutions, another Forth heavy weight player on the east coast. With a $1000 birthday gift from his wife, Don organized a one-day, project-oriented session with Chuck Moore, Bill Ragsdale and a chip designer to discuss the feasibility of building a Forth engine on silicon. The discussions affirmed for Chuck that his dream of Forth chip could be realized and that others were serious in their support of his idea.

John Peers saw the beauty of approaching Forth from several levels. He founded Technology Industries in March 1981, to be a parent organization to develop Forth in hardware, software, and applications. With funding from the Technology Industries, Chuck developed and demonstrated a Forth engine simulator with color CRT display of the internal data paths and operations in March 1983. A funding partner of the technology Industries, Sysorex International, became interested in the project and formed the Novix partnership in March 1984 to carry forward the hardware implementation of the Forth chip.

Mostek was chosen as the foundary to cast Forth in silicon. The chip is implemented with the 3 micron HCMOS process using 4000 gates. It was packaged in a 128 pin pin-grid array. The first working chip was delivered in March 1985, running at 8 MHz. It was supposed to be a prototype chip. Since 95% of the functions worked as designed, Novix decided to offer it as a product and called it NC4000P. An evaluation and development system named Beta Board was also offered by Novix.

As Mostek was sold to Thomson, and Novix went through a reorganization and incorporation process, efforts in removing the bugs in the prototype mask was suspended. A second run using the same prototype mask

was produced, with the pin count on the pin-grid package reduced to 121 pins. It seems that this prototype chip, complete with all the bugs and restrictions, will be the one we have to live with for a while. Even with these bugs and restrictions, NC4000P is more powerful than the best of the 16-bit microprocessors built by very reputable manufacturers.

As of March 1986, Novix has incorporated with Mr. John Peers as its chairman and chief executive officer. It is producing NC4000 chips and selling them. Chuck Moore is offering the Gamma Board, a kit with a bare PC board, a NC4000 chip and a pair of PROM--to people who like to build computers. Software Composers are selling board level products: SC1000C single board computer with NC4000 and 8K of RAM, prototyping PC boards, memory expansion boards, etc. As the chip becomes available, we shall see more products and manufacturers using this chip in various fields for different applications.

Forth didn't become a household name in computer circles until Forth Interest Group was formed in the San Francisco Bay area, which distributed fig-Forth source code by the ton at cost, beginning in 1978. Although Forth had established a sizable following in the microcomputer user community, the microcomputer industry was very slow in accepting Forth as an architecture for hardware design and implementation. There were scattered efforts toward building Forth engines using bit slice technology and random logic, but the consequence was almost nil. Meanwhile, Chuck became restive and took it upon himself the task of casting Forth into silicon. In 1980, Chuck left Forth, Inc. to pursuit his new dream.

In the reorganization, Forth, Inc. expanded its board of directors to include Bill Ragsdale, the founding father of Forth Interest Group, and John Peers of Logical Machines, Both of them had very intense interest in seeing Forth put into hardware. The right mix of personality was thus gathered for the precise chemistry necessary for brewing a Forth chip.

At that time, Chuck was designing a prototype board which would execute Forth words as primitive instructions with Glenn Haydon and others. The spark that initiate the Forth chip development was set by Don Colburn of Creative Solutions, another Forth heavy weight player on the east coast. With a $1000 birthday gift from his wife, Don organized a one-day, project-oriented session with Chuck Moore, Bill Ragsdale and a chip designer to discuss the feasibility of building a Forth engine on silicon. The discussions affirmed for Chuck that his dream of Forth chip could be realized and that others were serious in their support of his idea.

John Peers saw the beauty of approaching Forth from several levels. He founded Technology Industries in March 1981, to be a parent organization to develop Forth in hardware, software, and applications. With funding from the Technology Industries, Chuck developed and demonstrated a Forth engine simulator with color CRT display of the internal data paths and operations in March 1983. A funding partner of the technology Industries, Sysorex International, became interested in the project and formed the Novix partnership in March 1984 to carry forward the hardware implementation of the Forth chip.

Mostek was chosen as the foundary to cast Forth in silicon. The chip is implemented with the 3 micron HCMOS process using 4000 gates. It was packaged in a 128 pin pin-grid array. The first working chip was delivered in March 1985, running at 8 MHz. It was supposed to be a prototype chip. Since 95% of the functions worked as designed, Novix decided to offer it as a product and called it NC4000P. An evaluation and development system named Beta Board was also offered by Novix.

As Mostek was sold to Thomson, and Novix went through a reorganization and incorporation process, efforts in removing the bugs in the prototype mask was suspended. A second run using the same prototype mask was produced, with the pin count on the pin-grid package reduced to 121 pins. It seems that this prototype chip, complete with all the bugs and restrictions, will be the one we have to live with for a while. Even with

these bugs and restrictions, NC4000P is more powerful than the best of the 16-bit microprocessors built by very reputable manufacturers.

As of March 1986, Novix has incorporated with Mr. John Peers as its chairman and chief executive officer. It is producing NC4000 chips and selling them. Chuck Moore is offering the Gamma Board, a kit with a bare PC board, a NC4000 chip and a pair of PROM--to people who like to build computers. Software Composers are selling board level products: SC1000C single board computer with NC4000 and 8K of RAM, prototyping PC boards, memory expansion boards, etc. As the chip becomes available, we shall see more products and manufacturers using this chip in various fields for different applications.

2. Features of NC4000 Chip

The Novix NC4000 is a super high-speed processing engine which is designed to directly execute high level Forth instructions. The single chip microprocessor, NC4000, gains its remarkable performance by eliminating both the ordinary assembly language and internal microcode which, in most conventional processors, intervene between the high level application and the hardware.

A number of distinguishing features of this Forth engine on silicon are summarized as follows:

- 16 bit high speed, HCMOS single chip Forth engine.
- Direct execution of most Forth primitives in a single machine cycle without internal microcode.
- One cycle subroutine calls with mostly zero cycle returns.
- Supports 64K word memory, or 4M bytes with address extension port (the X-port).
- Fully static operation permiting very low power consumption suitable for battery powered applications.
- One cycle structured IF, ELSE, and LOOP instructions.
- Multiplication, division, and square-root step instructions.
- TIMES instruction allowing any instruction, including auto-incrementing/decrementing memory access, to be repeated once per cycle.
- single instruction fetch and store from/to the local memory.
- One cycle generation of hex FFFF.
- 257 element 16 bit hardware return stack with the top element in an on-chip I register.
- 258 element 16 bit hardware data stack with top two elements in on-chip T and N registers.
- Two versatile I/O ports, both of which are bidirectional, maskable, auto-comparable, and programmable for either latched or tristate output.
- Simultaneous access of return stack, data stack, main memory, and I/O port, concurrent with operation of ALU and shifter.
- Execution of multiple Forth words in a single cycle instruction, e.g. "OVER + ;", yielding over 180 available instruction combinations, not including permutations of register addressing.

3. External Data Paths

NC4000 chip is housed in a 121 pin pin-grid package. The pin layout is show is Fig. 1.1. The names and function of the pin groups are shown in Table 1.1.

The external data paths spawn by the large number of pins can be shown schematically in Fig. 1.2. The pins can be grouped into five different functional groups: Main memory data and address, data stack data and address, return stack data and address, I/O ports, and timing/control.

```
Table 1.1.   NC4000 Pin Names and Functions.
```

A0-A15	Main Memory Address Bus
B0-B15	I/O Port Bus
CLK	Processor Clock Input
INT	External Interrupt
J0-J7	Return Stack Address Bus
K0-K7	Data Stack Address Bus
D0-D15	Main Memory Data Bus
R0-R15	Return Stack Data Bus
RST	Processor Reset
S0-S15	Data Stack Data Bus
VDD	Power Supply
VSS	Ground
WEB	I/O Port Write Enable
WED	Main Memory Write Enable
WER	Return Stack Write Enable
WES	Data Stack Write Enable
X0-X4	Address Extension Port

```
Fig. 1.1.   NC4000 Pin Layout.

Fig. 1.2.   External Data Paths of NC4000.
```

3.1. Main Memory

The NC4000 controls and communicates with the main memory through 16 address lines, 16 data lines, and a write-enable line WED. The memory addressing space is thus 64K words or 128K bytes. The timing of the memory is synchronized by a single phase clock signal. At the rising edge of the clock, data from the main memory is latched into the data memory port. At the failing edge of the clock, memory address lines are stablized and addresses are available. The main memory must put the requested data on the data lines before the rising edge of the clock. The speed of the clock is thus dependent on the time required by the NC4000 chip to calculate the next address, which determines the length of the high period of the clock, and the time required by the memory to put valid data on the data lines, which determines the length of the low period of the clock. The high period, as required by NC4000 is 65 ns at the minimum, and the low period depends on the memory used in the system. Using high speed CMOS RAM with 50 ns access time, the clock speed can be pushed to about 8 MHz. Using low cost CMOS static RAM with 150-200 ns access time, 4 MHz would be more appropriate.

There are a few restrictions on the use of memory. Although the chip can address 64K words of memory, only the lower 32K can be used as program memory because the MSB bit of an instruction is used to indicate a subroutine call. However, the top 32K words can be addressed as data memory. Since the hardware reset causes the chip to start executing the instruction located at memory location 1000H, it is mandatory that the bootstrap routine be programmed into this and the subsequent memory. Thus ROM memory must occupy a

block of memory space starting at 1000H. Memory location 0 to 31 are special, in that these memory locations can be accessed by the NC4000 with single word instructions, while other memory locations can only be accessed by two word instructions. Hence the memory starting at 0 is preferrably RAM memory if the software is able to take advantage of this hardware function.

Whenever new data are to be written to the main memory, the WED (memory write enable) line will be brought low to coincide with the low period of the system clock. This line should be tied with the write enable lines of the RAM memory so that new data can be written into RAM memory.

This chip is intended to operate with static RAM memory chips that do not require a complicated memory refresh process.

The memory space can be greatly enlarged if the 5 I/O lines of the X-port (Extension Port) are used as extra address lines to control the main memory. In this manner the addressable memory can be expanded to 2M words or 4M bytes.

3.2. The Data stack and the Return stack.

A Forth engines requires at least two stacks, one to store return addresses for unfinished subroutines and the other to store parameters passed between subroutines. Since the gate array with 4000 gates cannot support the necessary memory to host two stacks, the data and address lines of these two stacks are brought off the chip. Each stack uses 16 data lines, 8 address lines, and a write enable line. Since the address lines are only 8 bit wide, the depth of the stacks is limited to 256 words in the external stack memory. If more than 256 words are pushed on to the external stack, the stack would wrap around like a circular buffer, and the data stored 256 words before the the current word would be lost.

The depth of the stack, does not seem to be that restrictive because most programs use a depth of only 10 words each, on the return or data stacks. The depth of the stacks will be a serious concern only when a recursive procedure is used. There, one has to be careful that the depth of the stacks is not exceeded.

The timing requirements for the stacks, are almost identical to those of the main memory. Stack data are latched into the chip when the system clock makes a low to high transition. The addresses to the stacks are stablized when the clock goes from high to low and the stack memory must put the data requested on the stack data lines before the clock goes from low to high. Thus the same type of memory used in the main memory can be used for stacks. There is little advantage to use higher speed memory for the stacks.

The write enable lines WES and WER to the stack memories are low for the low period of the system clock when data are to be written to the stack memory.

Since most commercial static CMOS memory chips have capacity larger than 256 bytes, it seems to be rather wasteful to use these chips for the stack without being able to fully utilize their capacity. One way to take advantage of the extra stack space is to use the lines in the X-port to perform bank switching of the stacks. This is very useful in supporting a multi-task system, in which each task has its own data and return stacks. Task switching in this environment will be extremely fast since the operating conditions of each task are fully preserved in their individual stack space.

3.3. B-Port and X-Port

Two input/output ports are supported by the NC4000 chip: a 16 bit B-port (B for bus) and a 5 bit X-port (X for extension). These two ports are fully programmable by the user through 4 internal registers for each port: a

direction register to specify individual bits to be input or output, a mask register to protect individual bits from being written to, a tristate register to tristate output bits, and a data register to read from pins and write to pins. Both ports can do I/O operations in single machine cycles as data registers are read or written. These 21 programmable, high speed I/O lines make NC4000 chip an extremely versatile controller chip for all types of high throughput, real time control applications.

The B-port write enable (WEB) line is low for the duration of the low period of the system clock when the output of the I/O ports is stablized. Data on the input lines are latched into the data register at the rising edge of the system clock as usual. Output data are available on the output pins about 100 ns after the rising edge of the system clock.

An interesting behavior of the I/O port is that a set of output data latches in the data register can be written to even when the bits are assigned as input. The data on the input pins are XOR with the bits in the output latches when read by the CPU. If the data latch were loaded with ones, we can invert the input data on the fly as they are read through the data register. This extra logic operation is programmable for each individual I/O pin.

3.4. System Timing and Control

The NC4000 is an asynchronous CPU chip which requires a single phase master clock. All internal registers are static and the information held in them remain indefinitely while the clock is held low. The higher limit of clock rate is set by the time required by the internal logic to calculate the address of the next instruction during the high period of the clock, which is about 65 ns in the prototype chip. Using a symmetric crystal oscillator for the master clock, this limits the speed of NC4000P to about 8 MHz. The low period of the clock is used mainly to wait for the external memory to put their date on the data lines. If one uses slower memory chips, the low period of the clock must be stretched accordingly.

The external clock signal is brought in via the CLK line. As long as the low and high periods satisfy the above requirements, the rate and the duty cycle of the clock are not critical. Either crystal oscillator or simple RC timing circuits can be used to generate the system clock signal.

The reset signal RST if brought low will stop all internal operations in the chip. When RST is brought back to high, NC4000 will jump to the reset memory location at 1000H and start executing the instruction fetched from that location. The software bootstrap routine must be placed in that location for the system to work properly.

There is also an interrupt input pin named INT. When the INT pin is brought low, NC4000 will execute a call instruction to location 20H, where a service subroutine must be placed. In the prototype chip, the use of this interrupt facility is severelly restricted because the interrupt can be serviced correctly only when a single cycle instruction is being executed. The interrupt will lose its return address if it occurs during the first cycle of a two cycle instruction. If precaution is taken to enable the interrupt only during a sequence of single cycle instructions, interrupt can be serviced correctly.

Being a CMOS gate array, the power supply voltage to the VDD pins can range from 0 to 7 volts. Nominal operation voltage is 5.0 Volts and typical current during operation is 10 mA. The supply voltage might have to be higher than 5 volts if it is to be used with a higher clock rate of 7 MHz or higher.

```
    Fig. 1.3.   Timing Diagrams of NC4000.
```

5. The Instruction Set of NC4000

Let's think of how a conventional computer interacts with its users and the layers the computer has to go through between accepting a command from the user and actually performing a function inside the computer. The command is issued to the operating system, which calls a compiled program into memory for execution. The program would usually be constructed from a set of statements written in either a high level language or in assembly language. This program would have to be compiled or assembled into a set of machine instructions. When the machine instructions are executed, microcode inside the CPU are invoked which actually do the dirty work of operating the gates and shuffling the bits. Thus there are at least 5 levels of interpretation. The NC4000 architecture reduces this type of interpretation to only two levels, user commands and machine instructions. The greatly reduced complexity between the user and the actual action of the computer is the most important reason for the speed and the versatility of NC4000 chip.

NC4000 is a 16 bit microprocessor. Its basic data elements and instructions are all in 16 bit words or cells. The instructions are sometimes called 'external microcode' in the sense that the instruction decoder would take individual bits in the instruction and perform individually assigned functions in parallel. It does not need another layer of microcode to perform the designed functions of an instruction. The side benefits are that many Forth instructions can be performed in a single machine cycle.

There has been many projects implementing Forth engines in hardware. All of these designs have attempted to encode individual Forth words into single machine instructions. They were shown to be much faster than Forth engines implemented in software because of the reduced overhead in NEXT, NEST, and UNNEST instructions and in the operation of the stacks. Chuck Moore went far beyond them in NC4000. What he attempted was to find the simplest way to controll the stacks and perform the operations while only using the top-most elements on the stack. He discovered that many of these functions can be performed independent of one another. Pushing or popping the stacks, arithmetic/logic operations, accessing main memory, and input/output are operations in different domains of the microprocessor. They do not have to be performed serially. These distinct, almost independent domains can be controlled by the very limited number of bits in a 16 bit instruction similar to bits in microcode. Thus it is possible to perform many functions in a single machine cycle, instead of using many machine cycles to perform one function, such as in all conventional microcode based microprocessors.

.new

5.1. Classes of NC4000 Instructions

There are four major classes of instructions in NC4000:

the subroutine calls, the I/O and memory instructions, the branch and loop instructions and the ALU instructions. The class of any instruction is decoded in the most significant four bits of the machine instruction, as shown in Fig. 1.5.

Fig. 1.5. Decoding of NC4000 Instructions.

```
Call                        0 a a a a a a a a a a a a a a a
ALU                         1 0 0 0 x x x x x x x x x x x x
Branch                      1 0 b b a a a a a a a a a a a a
```

```
Memory                      1 1 x x x x x x x x l l l l l l
       a: address    b: branch     l: literal    x: control
```

Bit 15 is truly the most significant bit in the NC4000 machine instruction. If it is zero, the instruction is a subroutine call and the rest of the instruction contains a 15 bit subroutine address. Zero in this bit position triggers the subroutine threading mechanism in NC4000. The program counter is pushed onto the return stack, i.e., copied into the I register. The 15 bit address in the instruction is moved into the address multiplexer A. At the beginning of the next machine cycle, the instruction stored at that address will be fetched for execution. The program counter P will be pointing at the next instruction in that subroutine.

Using bit 15 to encode a subroutine call has only one drawback--it can only call subroutines in the lower 32K word memory. The upper 32K word addressible memory cannot be used to store executable programs. It was a very serious trade-off in the design of the chip. The most important argument to support this trade-off is that Forth programs written for NC4000 can be extremely compact because of the single cycle subroutine calls and the condensation of many functions into a single instruction. Many large programs are needed to fill up the 32K word program memory. By the time memory requirements exceed 32K words, we will probably have a 32 bit Novix chip to take care of those stupendous programs that result from lazy-minds or uncommunicative programming teams.

When bit 15 is set, bit 14 is used to distinguish IO/memory instructions from ALU/branch instructions. When bit 14 is zero, the next two bits are used to decide whether the rest of the instruction is to be decoded as an ALU instruction or used as a 12 bit branch address. When bit 14 is one, then the rest of the instruction will be decoded to determine the type of I/O or memory instruction and how the I/O or memory is to be accessed.

```
    Fig. 1.6.   NC4000 Instruction Formats
```

```
Call                        0          address
ALU                         1 0 0 0  ALU   Y Tn ; SA D %SLSR
IF (conditional branch)     1 0 0 1        address
LOOP                        1 0 1 0        address
ELSE (unconditional branch) 1 0 1 1        address
Literal Fetch               1 1 0 0  ALU   Y Tn ;  literal
Literal Store               1 1 0 1  ALU   Y Tn ;  literal
Memory Fetch                1 1 1 0  ALU   Y Tn ;  literal
Memory Store                1 1 1 1  ALU   Y Tn ;  literal
```

Another way of classifying the NC4000 instruction set is shown in Fig. 1.6. In this figure, the non-subroutine call instructions are classified according to the instruction type field, bits 12-14. In this figure, all the bits which perform specific functions are named and placed in their respective bit positions. We shall discuss each of these instruction types in great detail in later sections. Only a few general comments will be made here, as an overview of this instruction set.

In all I/O, memory, and ALU instructions, types 4-6 and type 0, bit 5 is called the return bit. This bit, when set, will cause a subroutine return, in addition to whatever the instruction may otherwise do. Therefore, a subroutine return in NC4000 can be a zero cycle operation, since it gets a free ride if the last instruction in a

subroutine is an IO/memory or ALU instruction. It would be very difficult to optimize a subroutine return instruction any further.

In the prototype NC4000P chip, however, this return bit should not be tagged to a two cycle memory instruction, because the return operation will interfere with the memory fetching. The memory address will be replaced by the return address from the return stack. This is not a problem with single cycle memory instructions as there is no conflict in the use of the address multiplexer.

In IO/memory and ALU instructions, bits 9 to 11 in the ALU field determines the function of the ALU section on the chip. Thus a free ALU operation can be tagged on to an IO/memory operation.

An ALU operation requires two operands. One of the operands is always taken from the Top register T and the other operand is usually selected from an internal register, specified by a two bit field Y, bits 7 and 8. In most instances, the Y field is zero, which selects the Next register N as its operand.

External data stacks are controlled by two bits: Tn bit at bit 6 and SA bit at bit 4 in an ALU instruction, or bit 14 in a memory instruction. Tn bit, if set, copies the contents of T into N at the beginning of an instruction. The SA, stack active bit, signals the external stack to perform a push or a pop operation. If both Tn and SA bits are set, old T register is copied into N register and the contents of the N register is pushed on the external data stack. If Tn is zero and SA is set, then the top element on the data stack is popped back into the N register and the data in N register is lost. The combination of control bits in the ALU field, the Y selector, and the Tn and SA bits allow the NC4000 to perform most Forth ALU operations and stack operations, and as well as combinations of the ALU and stack operations.

In a memory instruction, the least significant 5 bits constitute a literal field, which contains a small integer from 0 to 31. This literal is used to represent different types of information needed by the memory instruction. In a short literal instruction, the small literal is pushed on the data stack as an integer. In an internal register accessing instruction, it selects one of the registers to be accessed. In a local memory instruction, it is the address of a local memory (the first 32 words in the main memory). These local memory words can thus be accessed by a single instruction. In the extended memory instructions, it supplies the page number to be put out on the X-port, to select one of the 32 memory pages for the memory instruction to access. In many instructions, this field is not needed and must be cleared to zero.

In a branch instruction, bits 12 and 13 determine which type of branch it is and the lower 12 bits supply the target address. The 12 bit address field specifies an absolute address within the 4K word page which contains the branch instruction. It is thus impossible to branch across a 4K word boundary. The programmer must be aware of this property in the branch instructions when he is close to a page boundary.

A 1 in the two bit field (bits 12 to 13) indicates an IF instruction, which does a conditional branch to the following 12 bit address. The branch condition is taken from the Top register T. If T register is zero, the IF instruction becomes a DROP. A 3 in this field indicates an unconditional branch or an ELSE instruction. The 12 bit address in the address field is always taken as the address of the next instruction. If a 2 is in this two bit field, the instruction is a NEXT instruction, which will decrement the I register--containing the loop index--and will branch to the 12 bit address if I is not 0. When I is decremented to zero, the conditional end of loop, the NEXT instruction pops the index off the return stack and terminates the loop.

The last comment about this rather complicated instruction set in NC4000 is that not all combinations of control bits can generate meaningful instructions. Certain combinations simply do not make sense at all and other combinations cause conflicting use of the registers or data paths and the results are not always

predictable. In addition, defects in the prototype NC4000P precludes some instruction or combinations of bits and those instructions should not be used. In Table 1.7-8, we have collected all the valid ALU instructions and instruction combinations that can be safely used in the NC4000P chip. Many of these restriction will be removed when the production chip becomes available.

5.2. The ALU Instructions

ALU instructions are the most complicated class of instructions in the NC4000 chip because all of the 12 lower bits in the instructions are decoded to perform simultaneous functions. A firm grasp on the use of the individual bits and their interactions is essential in understanding the inner mechanism of this chip. Understanding will lead to an appreciation for the power of these instructions which can compress several Forth words into one machine instruction. What we hope to do in this section is to go through each field and their function A number of examples will also be given to illustrate how the field and bits can be combine to form multiple function instructions. With this information and examples, you will probably be able to decode other valid instructions and to visualize how they would work.

A more detailed data path diagram for the ALU section of the NC4000 can be of great help in explaining the inner mechanism of the ALU instructions, as shown in Fig. 1.7.

The ALU performs arithmetic and logic operations using operands supplied to it from the T register and the Y port. The function of the ALU is specified in the ALU field in the instruction, bits 9-11. 8 different actions can be performed on the two operands as shown in Table 1.3.

```
Table 1.3.   ALU Code and Function
```

ALU Code	Function
0	Pass T
1	T AND Y
2	T - Y
3	T OR Y
4	T + Y
5	T XOR Y
6	Y - T
7	Pass Y

.new .5 Fig. 1.7. Data Paths and Registers in the ALU Section .new .3

The two bit Y field, bits 7 and 8, controls the multiplexer which selects one of four registers as the source for the Y port as in Table 1.4.

```
Table 1.4.   Y-Port Selector
```

Y Code	Source to Y Port
0	N register

```
1                   N register with carry
2                   MD Multiplier/divisor register
3                   SR Square-root register
```

Normal operations use the N register as the Y-port operand. The N register with carry is selected when doing extended precision arithmetics. MD register is used to store multiplier in multiplication operations or divisor in division. Both the MD and SR registers are involved when a square-root operation is performed.

Tn, bit 6, operates a switch which connects the N register to the output of T register. The contents of the T register is copied into the N register at the beginning of an ALU machine cycle if Tn bit is set. If T is passed through the ALU unit unchanged and Tn bit is set, the net result is a DUP operation. If Tn is 0 in this case, the net result will be a NOP operation. If N is selected as the input to the ALU unit and passed through it unchanged, a DROP DUP operation will be performed when Tn is 0 and a SWAP will be performed when Tn is 1.

The Stack Active bit SA, bit 4 in the instruction, activates the external data stack in the sense that either the contents of N register is pushed on the external stack or the top element on the external stack is popped into N register. However, the exact involvement of the external stack in this instruction also depends upon the state of Tn bit. The stack action can be summarized in Table 1.5. .new

```
Table 1.5.   Data Stack Code and Functions
```

```
SA      Tn          Stack Function
0       0           ALU result to T.  N not changed.
                      No stack action.
0       1           ALU result to T.  Old T to N.  No
                    stack action.
1       0           ALU result to T.  External stack
                    popped into N register.
1       1           ALU result to T.  Old T to N.  Old
                    N pushed on external stack.
```

It would be more pleasing if an independent bit were assigned to specify the direction of the stack activity besides the Tn bit. However, it was found that the limited combinations of these two bits are sufficient to implement most of the stack operations required by the Forth language, while many more can be synthesized in conjunction with other activities in the ALU section. The fact that these two bits are not in a contiguous field sometime makes it difficult to associate the instructions with their stack effects. These were the trade-offs the designer had to make and the users have to live with them.

Bit 5 is the almighty return bit. When this bit is set, a return from subroutine function will be triggered even as the ALU and stack functions are performed concurrently. The return bit undoes the subroutine call, as executed by the Call instruction. The return address on top of the return stack or the I register is popped into the address multiplexer A. The next instruction executed will be the instruction following the Call instruction and the execution sequence will resume. The return bit can be tagged to any ALU instruction. Thereby a free

return is generated without an explicit return instruction. If this return bit is zero, execution will continue with the next instruction whose address is in the program counter P.

At the bottom of the ALU unit, there is a shifter which can shift the results from the ALU right or left by one bit before storing the results into the T register. The shifter is controlled by the three LSB bits in an ALU instruction:D, SL, and SR bits, or bits 3, 1 and 0, respectively. The bit patterns and their functions are shown in Table 1.6. .new

```
Table 1.6.   Shift Code and Function
```

```
D    SL   SR          Function
0      0     0           No shift.
0      0     1           16 bit shift right.
0      1     0           16 bit shift left.
0      1     1           Sign extension of N into T.
1      0     0           Not valid.
1      0     1           32 bit shift right.
1      1     0           32 bit shift left.
1      1     1           Not valid.
```

The bit pattern 100 and 111 above, are not valid in the prototype chip. The designed function of the 100 pattern is to shift the N register left one bit. The designed function of the 111 pattern is to shift the N register right by one bit and extend its sign into the T register. Chip defects in the prototype cause these functions to behave erratically.

Bit 2, the % or divide bit, is used only in the three divide instructions: the divide step /', the last divide step /", and the square-root step S'. When it is set, a conditional subtraction is performed. If the subtraction does not generate a carry, the difference is passed to the T register. If a carry is generated, meaning that the divide step should not be performed, the results of subtraction is not written to T register. Division and square root can be implemented by these conditional subtraction steps.

These discussions complete the description of the fields and bits in the ALU instructions and their functions. Because so many thing can happen simultaneously, it is rather difficult to completely understand this ALU section and the different instructions the chip can perform. The best we can hope to do is to take some of the valid ALU instructions and analyze them to familiarize yourself with the instruction set. On the other hand, many of the combinations of functions can be automatically resolved by an optimizing compiler. It can be made to recognize permissible and restricted Forth word sequences and compile the most compact machine instructions to fully utilize the power of the NC4000 chip. A better understanding of the inner mechanism of this ALU would enable the user to anticipate the optimization process and thus assure the production of the most efficient code. .new

```
Table 1.7.   Valid ALU Instructions
```

```
Code     a=7 (Pass Y)        a=0 (Pass T)        a=Arith/Logic
10a000   DROP DUP            NOP                 OVER a-
10a001   ---                 2/                  OVER a- 2/
10a002   ---                 2*                  OVER a- 2*
10a003   ---                 0<                  ---
10a010   ---                 ---                 ---
10a011   ---                 D2/                 ---
10a012   ---                 D2*                 ---
10a013   ---                 ---                 ---
10a020   DROP                NIP                 a
10a021   ---                 2/ NIP              a 2/
10a022   ---                 2* NIP              a 2*
10a023   ---                 NIP 0<              ---
10a030   ---                 ---                 ---
10a031   ---                 ---                 ---
10a032   ---                 ---                 ---
10a033   ---                 ---                 ---
10a100   SWAP                NIP DUP             SWAP OVER a
10a101   ---                 NIP DUP 2/          SWAP OVER a 2/
10a102   ---                 NIP DUP 2*          SWAP OVER a 2*
10a103   ---                 NIP DUP 0<          ---
10a110   ---                 ---                 ---
10a111   ---                 ---                 ---
10a112   ---                 ---                 ---
10a113   ---                 ---                 ---
10a120   OVER                DUP                 2DUP a
10a121   ---                 DUP 2/              2DUP a 2/
10a122   ---                 DUP 2*              2DUP a 2*
10a123   ---                 DUP 0<              ---
10a130   ---                 ---                 ---
10a131   ---                 ---                 ---
10a132   ---                 ---                 ---
10a133   ---                 ---                 ---
```

```
     Special ALU Instructions
```

```
102411   *-                  102412   *F
102414   /"                  102416   /'
102616   S'                  104411   *'
```

5.3. I/O and Memory Instructions

The I/O and memory instructions in NC4000 are characterized by the fact that the two most significant bits in the instructions are set and that the least significant 5 bits often contain a literal value. The other 9 bits in between are decoded to perform ALU, data stack, and return operations. The general format of this class of instructions is as follows:

```
I/O and Memory Instructions   1 1 X !  ALU  Y CSA ;  literal
```

However, there are many special cases causing the I/O and memory instructions to appear as if they are governed by random logic. The best one can do is to present the entire table of valid I/O and memory instructions as shown in Table 1.8.

Based on the table, we can make a few observations which might guide the user in understanding these rather complicated instructions.

The store bit, bit 12, will always be controlling read and write to/from the memory or registers. If this bit is zero, the instruction is a fetch operation; otherwise, it must be a store instruction.

The ALU field is also always predictable, as it specifies what kind of ALU operation shall be performed on the operands. In most cases, the ALU operation can be performed on the operand while the I/O or memory operation is being processed. However, it is not always obvious as to which operands are used in the ALU operation.

Bit 6 is very close in function to the SA bit in the ALU instructions. If it is zero, the data stack depth is not changed and all operations are performed on the T and N registers. these two registers will contain the results when the instruction is completed. The C bit, bit 7, is similar to the lower bit in the Y field of the ALU instruction. It selects the N register as the input to the Y-port of the ALU unit. If it is set, the carry bit is also used in the ALU operation; otherwise, only the N register is used without carry. .new

```
    Table 1.8.   Valid I/O and Memory Instructions
```

```
Code     a=7 (Pass Y)        a=0 (Pass T)        a= Arith/Logic


14a0nn   ---                 ---                 nn @ a-
14a1nn   nn @                ---                 ---
14a2nn   ---                 ---                 nn @ c-
14a3nn   nn I@               ---                 ---

14a400   ---                 ---                 n a-
14a500   n                   ---                 ---
14a600   ---                 ---                 n c-
14a7nn   ---                 ---                 nn I@ a-

15a0nn   nn !                ---                 DUP nn ! a
15a1nn   ---                 ---                 ---
15a2nn   nn I!               ---                 DUP nn I! c
15a3nn   ---                 DUP nn I!           ---

15a4nn   ---                 ---                 nn a-
15a5nn   nn                  ---                 ---
15a6nn   ---                 ---                 nn c-
15a7nn   nn I@!              ---                 ---
```

```
16a000   ---                 ---                 @ a-
16a100   @                   ---                 ---
16a200   ---                 ---                 @ c-
16a300   ---                 ---                 ---

16a4nn   ---                 ---                 nn X@ a-
16a5nn   nn X@               ---                 ---
16a6nn   ---                 ---                 nn X@ c-
16a7nn   ---                 ---                 nn @a-

17a000   !                   ---                 ---
17a100   ---                 ---                 ---
17a200   ---                 ---                 ---
17a300   OVER SWAP !         ---                 ---

17a4nn   nn X!               ---                 ---
17a5nn   DUP nn X!           ---                 ---
17a6nn   ---                 ---                 ---
17a7nn   ---                 ---                 nn !a-
```

```
     Special Return Stack Instructions
```

```
140721   R> DROP             157201   >R
147301   R@                  157221   TIMES
147321   R>                  157701   R> SWAP >R

         a-: SWAP -     c-: SWAP -c
```

.new

Bit 8 selects alternate ways of accessing different types of memory or I/O. In case of literal fetch instructions, setting bit 8 would cause a fetch of the 16 bit literal value following the instruction. A zero in bit 8 would fetch the short literal embedded in the instruction itself. Extended memory fetch and store instructions (16xxxx and 17xxxx types) are invoked by setting bit 8 to indicate that the extended memory addressing mode is selected.

When bits 6 to 9 are all set (1xx7xx type), the instruction refers to some internal registers, whose register number is encoded in the 5 bit literal field. There are 17 addressible registers in the NC4000. Their register numbers, assigned names and functions are listed in Table 1.9.

```
   Table 1.9.   NC4000 Internal Registers
```

```
       Name     Number       Function
       J/K              0             Data/Return Stack Pointers
```

```
I                  1              Return Index Register
P                 2              Program Counter
-1                3             True Register
MD             4/5           Multiplier/Divisor Register
SR               6/7           Square-root Register
B                   8            B-Port Data Register
B Mask          9            B-Port Mask Register
B I/O            10           B-Port Direction register
B Tristate      11           B-Port Tristate Register
X                  12           X-Port Data Register
X Mask         13           X-Port Mask Register
X I/O            14           X-Port Direction Register
X Tristate      15           X-Port Tristate Register
#Times          17           I as TIMES Counter
```

MD and SR registers are used for special purposes, i.e., to hold necessary parameters for doing more complicated arithmetic operations such as multiply, divide, and square-root. However, if they are not used by these specialized instructions, these registers are available for storing temporary data for convenient retrieval.

In performing I/O functions and communication with the outside world through the B-port and X-port, one only has to read or write the B or X data registers and the data will be read from the input pins or be written to the output pins. Before the actual I/O operations, the pins must be initialized and assigned appropriate functions. The function of each I/O pin can be programmed via the I/O, Mask, and Tristate Registers in the B- and X-ports. The exact function of bits in these registers are shown in the following table:

.new

```
Table 1.10.   Function of Bits in the I/O Registers
```

Register	Bit function if set	
	Input	Output
Data	Set comparison-latch	---
Mask	---	Inhibit writing
I/O	Set for output	Set for output
Tristate	---	Set to tristate after write cycle.

CHAPTER II. BUILDING A NC4000 COMPUTER

1. Commerical Products Using NC4000 Chip

One of the major design goals Chuck Moore wanted to achieve with the NC4000 chip was ease in constructing a high performance computer with minimal parts and resources. All the necessary signals are brought out to the pins on the chip package, making it very easy to attach external memory and control circuitry to form a complete computer system. There are several computers already being built and are available commercially, based on the NC4000 chip. Many others have been custom built to solve specific problems.

When the first wafer of NC4000 was diced at Mostek, about 80 good chips passed the functional tests. Chuck Moore was the first person to take delivery from this batch. He built a PC board to host these first chips and called the resulting computer 'Alpha Board' for obvious reasons. Chuck used the Alpha Board to demonstrate the performance of NC4000 and developed the first package of software for it. Alpha Board was Chuck's personal computer and was not distributed commercially. The board is about 6" by 4" in size, containing about 8K words of memory, 2K words for two stacks, a few glue chips, and a clock. It ran at 7 MHz maximum. Chuck was able to generate color CRT displays with this board in many of his demonstrations.

Novix took the rest of the batch from Mostek and built Beta Boards with them. The Beta Boards were sold by Novix as development tools for NC4000 chips. The boards measured 10" by 14". It used high speed RAM's and ROM's and was specified to run at 7-8 MHz. 64K words of memory are on board, as well as two RS-232 serial ports and a SCSI disk interface. The software delivered with the Beta Board was a version of poly-FORTH developed for Novix by Forth, Inc. There are two versions of the Beta Board system, one using an IBM PC as a host computer and the other a stand alone system with its own terminal and disk drives.

The second batch of NC4000 chips were made by Mostek and delivered to Novix in early 1986. This batch used the same mask as the first batch, but the pin out and the package were modified. Novix will build and sell Beta Boards using the new NC4000 chip. Novix is also advertising complete NC4000 development systems with hard disk drives and floppy disk drives using Beta Boards as the brain.

Chuck Moore updated the design of Alpha Board for the new NC4000 chips and is distributing a 'Gamma Board' kit, which consists of a NC4000 chip, a 6" by 4" PC board, and a pair of 2732's with the cmFORTH firmware. This kit is sold through his company Computer Cowboys in Woodside, California. The kit also contains 360 Augat Holtit press-fit sockets for mounting the chips. User must supply four 6264 CMOS memory chips, a 74HC132 NAND gate chip, a 4 MHz clock, and a few resistors and capacitors to populate the board. To use the board, one only has to supply 5 V to Vdd and hook a terminal to the pseudo RS-232 port on board.

Software Composers, a company in Palo Alto, California, is also building a single board computer based on NC4000 chip. This single board computer is called 'Delta Board' with a code name SC1000C. It is very similar to Gamma Board in design but with a different layout. The memory bus and the I/O bus are brought to a 72-finger edge connector with pertinent timing signals. The bus structure makes it easy to add memory and peripheral devices to the single board computer. It has 8K words of RAM, 4K of ROM, two stacks, and a serial port. It also uses cmFORTH as its operating system. The Delta Board is available both in assembled form or in kits. Software Composers is also producing supporting accessories such as power supply, back plane, expansion memory board, etc.

Since it is rather straightforward to use the NC4000 chip to build a computer, we will surely see more NC4000 products and more applications in the near future.

2. Build Your Own NC4000 Computer

What I want to do here is to describe a typical design of NC4000 based computer. By providing you with enough essential information on this design, you should be able to build a small computer using the NC4000 chip, or incorporate the design into your system to suit your application.

This design is very similar to that of the Gamma Board and that of the Delta Board, as Chuck Moore provided the basic information in helping us develop our prototype board. The schematics in the following sections are thus useful for users of either board. The design is broken down into three major sections: the CPU, the stacks, and the memory sections. Each section will be discussed in detail.

2.1. The CPU Section

The NC4000 chip, its timing, and I/O connections are shown in Fig. 2.1. The memory interface to the main memory and two stacks will be elaborated later. Here we shall only be concerned with the immediate control signals passed to the NC4000 chip.

The CLK input is driven by a single phase CMOS clock. The frequency of this clock depends on the memory speed and the maximum speed of the chip. A general requirement of the clock is that the high period of the clock must be longer than 65 ns to allow NC4000 enough time to generate correct memory and stack addresses. The low period of the system clock must be long enough for the memory to send data back on the data lines. If we wanted the chip to run at its full speed, all memory chips should have an access speed less than 60 ns. Using slower but cheaper memory chips with access time of about 100 ns, the maximum clock rate would be limited to less than 5 MHz. 4 MHz is a good choice with 150 ns memory chips.

A CMOS crystal clock provides stable and accurate timing for most applications. A simple RC oscillator can also be used as clock source. However, because the clock is used to control the baud rate of the RS-232 serial port, a CMOS crystal clock might be more appropriate. The duty cycle of the clock can vary from 40 to 60%.

The clock signal is distributed throughout the entire computer, synchronizing other components with NC4000. The low period of this clock is used to enable memory read or write. At the rising edge of the clock, most input signals to NC4000 are latched by NC4000. The memory and I/O write enable signals, i. e., WED, WER, WES and WEB, have different timing characteristics. However, when the clock is low, these enable signals are assured to be valid. The falling edge of the clock can thus be used to latch these enable signals into the memory or I/O device.

The RST (reset) input is connected through a NAND gate to an RC network. The RC network generates a reset sequence during power-up by holding RST input pin low for about 100 ms after 5 V power is applied to the chip. When RST is released to 5 Volts, NC4000 will execute the RESET word stored in ROM memory at address 1000H. The reset sequence assures that NC4000 is initialized properly and then enters into the text interpreter loop.

The INT (interrupt) input is normally pulled to 5 V through a pull-up resistor. When this input is grounded and then released to 5 V, an interrupt request flip-flop is set inside NC4000. If interrupt is enabled, NC4000 will make a subroutine call to memory location 32 where an interrupt service routine must reside. Return from this subroutine call will then reset the interrupt flip-flop for the next interrupt. When the interrupt request flip-flop is set by an external interrupt signal and the interrupt is disabled, the subroutine call to location 32 is suppressed but the flip-flop will remain set until interrupt is enabled and serviced. Further interrupts before the flip-flop is reset will then be lost.

Bit 8 (100H) in the Tristate Register of X-port (register 15) is the interrupt enable bit.

In the NC4000P prototype chip, the use of interrupt is severely limited because interrupt must not occur during a two cycle memory instruction or during a jump instruction. Interrupt will destroy the memory address in the address multiplexer and the interrupt service routine will lose its proper return address. Interrupt can only be enabled during a sequence of single cycle, non-jump instructions.

```
    Fig. 2.1.   CPU Section of a NC4000 Computer
```

2.2. I/O Ports

All sixteen B-port I/O lines and three of the X-port I/O lines--X1, x2 and X3-- are configured by the reset routine to be output lines and are pulled to ground. As a result it is safe to leave all these I/O lines open. Input lines to NC4000 must not be left floating because NC4000 tends to overheat if it finds unterminated inputs.

Each of the output lines thus configured can draw 16 mA from the device connected to it. If you are going to connect other devices to these ports, be sure that your device can withstand this abuse. To use any of these lines as input to NC4000, you will have to modify the RESET routine in cmFORTH so that NC4000 will be configured correctly upon power-up or reset.

X0 and X4 in the X-port are used to implement a serial communication port in this design. This serial port allows the NC4000 chip to talk to a standard RS-232 terminal. It is not a true RS-232 port because the voltage level is between 0 and 5 Volts. However, it does communicate correctly with all standard RS-232 equipment. X0 is the transmitter and X4 is the receiver. X0 can drive the receiver of a RS-232 device directly. X4 cannot be connected directly to a RS-232 transmitter because the transmitter swings to -9 volts. The negative swing must be limited to protect the diode in NC4000. The simplest solution is to put two 3K current limiting resistors between these two ports and the external RS-232 device.

2.3. Main Memory

In the design of a small computer with NC4000 as the central processing unit, there are two important constraints in the arrangement of memory. One is that the reset routine must begin at location 1000H and some ROM memory must be put in the neighborhood of 1000H for a self booting system. The other is that memory location 0 to 31 are local memory to NC4000, which can be accessed by single cell local memory instructions. cmFORTH uses many of these local memory cells to store system variables for fast access. Therefore, memory around location 0 must be RAM memory.

If memory chips came in 4K byte sizes, the memory design would be straightforward. We would decode memory in 4K pages and arrange ROM's and RAM's accordingly. However, low cost, static CMOS RAM's are available either in 2K or 8K byte sizes. The choice is either using many small chips or wasting space in large chips.

Chuck Moore suggested the following decoding scheme which would fully utilize a pair of 8K byte RAM chips with a pair of 4K byte PROM chips by partially decoding the RAM memory space. This decoding scheme is shown in Fig. 2.2. A12 address line is inverted by a NAND gate. The negated A12 signal is used to drive the positive chip select CE line of 6264 RAM chips and the negative output select ~OE of 2732 PROM chips.

Address line A13 is connected to the A12 pins on 6264 chips. This allows the RAM's to respond to addresses from 0 to 0FFFH and from 2000H to 2FFFH, while the PROM's reside between 1000H and 1FFFH.

This partial decoding method fully utilizes the 8K byte 6264 RAM chips. The problem is that it would not allow more than 8K words of RAM in the system. It is quite suitable for very small systems, but will make memory expansion very difficult.

For a system which must use more than 8K words of RAM memory, a conventional decoding scheme shown in Fig. 2.3 is more appropriate.

A 74HC138 1 of 8 decoder chip is used to select memory pages of 8K word size. Address lines A13, A14, and A15 generate address select signals to enable memory pages. In the lowest memory space or page 0, RAM must occupy addresses from 0 to FFFH and ROM must occupy addresses from 1000H to 1FFFH. This is achieved by using negated A12 to enable ROM via ~CE and using A12 to select RAM via ~OE. RAM chips above 2000H are selected by the 74138 decoder directly.

In this decoding method, half of the 8K RAM in page 0 is wasted. However, this system can accommodate 64K words of memory for a full blown NC4000 computer.

It is important that the chip select (CS) pins of the memory chips must always be enabled by tying them to ground, because the time delay in memory chips from chip select to data available is too long to be useed with NC4000. Using the chip enable (CE) and output enable (OE) to select appropriate chips allows these slow and inexpensive ROM and RAM chips to run at a rate much higher than that specified in the data sheets.

```
    Fig. 2.2.   Memory Decoding for a Small NC4000 Computer

    Fig. 2.3.   Memory Decoding for a Large NC4000 Computer
```

2.4. The Data Stack and the Return Stack

NC4000 supports two external stacks, one for subroutine return addresses and one for data to be passed between subroutines or words. Since these stacks have data paths independent of the main memory bus and I/O bus, NC4000 can access all these data buses simultaneously in a single clock cycle. The most significant result is that a subroutine call can be performed in a single clock cycle.

The data path to either stack includes a 16 bit wide data bus and an 8 bit wide address bus, in addition to the respective write enable line and the common clock signal. The 8 bit width of the address bus limits the depth of the external stacks to 256 words. For most application, two stacks of 256 words deep are more than adequate. However, it is difficult to find cheap static memory of this depth. Currently, the most readily available static RAM memory chips are either 2K bytes (6116) or 8K bytes (6264 or 6265). It seems to be a great waste to use only 256 bytes in them, but that's life.

In Fig. 2.4 the wiring of both the data and return stacks are shown schematically. We use 6264 chips as an example, because they are also used for the main memory. The circuit for smaller 6116 is almost identical and can be inferred easily. Using either type of RAM chips, the timing and control are similar. The chip select and output enable lines are always enabled by tying either to 5 V or to ground. The chip enable lines (~CE) are enabled by the main clock during the low half of the clock period. The write enable lines (WES and WER) are connected to the write enable lines (~WE) of the respective chips.

Since NC4000 only supplies 8 address lines to either stack, the extra address lines on the RAM chips must be either pulled to 5 V or grounded. If you absolutely must have more than 256 words for a stack, you can use the I/O lines in B-port or X-port to control the most significant address lines and swap the stacks in pages of 256 words.

```
    Fig. 2.4.   Data and Return Stacks of a NC4000 Computer
```

3. Circuit Board for NC4000 Computer

From the above description, a single board computer using NC4000 as its central processing unit is very simple, with a chip count of about 10. A 6" by 4" PC board is more than enough to host these chips. For those who can handle wirewrap guns or tools comfortably, constructing such a computer will be a one-day project. To avoid wiring errors, a printed circuit board is a much better way to go. Since both Chuck Moore's Computer Cowboys and Software Composers are supplying PC boards with the NC4000 chip, it is worth the extra costs to buy their kits and do the assembly yourself. You can order the kits Yrom from them directly:

```
        Novix Inc.
        10590 N. Tantau Ave.
        Cupertino, CA  95014
        (408) 253-6930

        Computer Cowboys
        410 Star Hill Road
        Woodside, CA  94062
        (415) 851-4362

        Software Composers
        210 California St., Suite F
        Palo Alto, CA  94306
        (415) 322-8763
```

A final note on the power supply. The NC4000 computer as described consumes about 200 mA at 5 V, with a 4 MHz clock. The voltage of the power supply is not very critical. It can vary from 4 to 6 Volts. A small regulated 5 V power supply of any kind is adequate. A wall mount 6 Volt power supply for video games should work well, too. Chuck tried the ultimate power supply: 3 or 4 alkaline D cells. He estimated that 3 D cells could last 30 hours and 4 cells, 50 hours. If you had a Radio Shack 100 portable computer to substitute for a terminal, you would have a truly portable and powerful computer in a briefcase.

CHAPTER III. THE cmFORTH OPERATING SYSTEM

This version of Forth, cmFORTH, was developed by Chuck Moore, the chief architect of the Novix-4000 Forth engine and the inventer of the Forth language. It is used in Chuck's Gamma Board and in Software Composers' SC1000C single board computer as their operating system. Chuck Moore and Novix kindly donated this remarkable software package into the public domain to encourage users exploring the phenominal capabilities of the NC4000 chip.

This chapter is intended to be the documentation for cmFORTH. I will attempt to go through the system in minute detail, in order to help those who are unfamiliar with the Forth language as well as those who are not familiar with Chuck Moore's coding style. In any case, the source listing itself is the primary documentation and the descriptions in this chapter are at best commentary to the source listing. The complete source listing of cmFORTH is included in this book as Appendix A. A program to burn target compiled object code into EPROM's is in Appendix B. A short program which must be implemented in a host computer to act as a terminal/disk server for the NC4000 computer is in Appendix C.

The original source listing of cmFORTH is your best source of code examples and programming style, when striving to use the NC4000 chip most effectively. You are encouraged to study this listing very carefully in order to gain maximum benefit from the chip. The source code is only 30 screens long. It would not be very difficult to study it and to be thoroughly familiar with it.

There are many important innovations in cmFORTH worthy of your special attention. First is the optimizing compiler which takes normal Forth source code and compiles very short and efficient NC4000 machine instructions. Next is the dual threaded vocabulary structure in which all the compiler directives and smart compiling/assembling words are linked together, separate from the regular FORTH vocabulary. The interpreter searches only the FORTH vocabulary. The compiler first searches the COMPILER vocabulary and then the FORTH vocabulary. This searching method eliminates the necessity of immediate words. Lastly, a very simple linking method is used to compile words into a target dictionary, which can later be isolated from the main dictionary and burned into EPROM. These innovations in time will make significant impact in the Forth community, leading to better and more efficient Forth systems and programs.

このForthのバージョンであるcmFORTHは、Novix-4000 Forthエンジンのチーフアーキテクトであり、Forth言語の発明者であるChuck Moore氏によって開発されました。 ChuckのGamma BoardやSoftware ComposersのSC1000Cシングルボードコンピュータのオペレーティングシステムとして使用されています。 Chuck MooreとNovixは、NC4000チップの驚異的な能力を探求するユーザーを奨励するために、この驚くべきソフトウェアパッケージをパブリックドメインに寄贈してくれました。

この章は、cmFORTHのドキュメントとなることを意図しています。 Forth言語に不慣れな人や、Chuck Mooreのコーディングスタイルに馴染みのない人の助けになるように、システムの細部まで説明することを試みます。 いずれにせよ、ソースリストそのものが主要な文書であり、本章の記述はせいぜいソースリストの解説に過ぎない。 cmFORTHの完全なソースリストは、本書に付録Aとして収録されています。 ターゲットコンパイルされたオブジェクトコードをEPROMに書き込むプログラムは、付録Bにあります。 NC4000 コンピュータのターミナル/ディスクサーバーとして動作するために、ホストコンピュータに実装されなければならない短いプログラムは、付録Cにあります。

cmFORTHのオリジナルのソースリストは、NC4000チップを最も効果的に使用しようとするとき、コード例とプログラミングスタイルの最良の情報源となります。 このチップから最大限の利益を得るために、このリストを非常に注意深く研究することが推奨されます。 ソースコードの長さはわずか30画面です。 それを勉強し、徹底的に熟知することは、それほど難しいことではないでしょう。

cmFORTHには、特別な注意を払うに値する多くの重要な革新的技術があります。 まず、最適化コンパイラが、通常のForthソースコードから、非常に短く効率的なNC4000マシン命令をコンパイルします。 次に、デュアルスレッド語彙構造で、すべてのコンパイラ指令とスマートコンパイル/アセンブル語は、通常のFORTH語彙とは別に、一緒にリンクされています。 インタプリタは、FORTHの語彙だけを検索します。 コンパイラは、まず COMPILER 語彙を検索し、次に FORTH 語彙を検索します。 この検索方法によって、直前の単語が不要になります。 最後に、非常に簡単なリンク方式で、単語をターゲット辞書にコンパイルし、後にメイ

ン辞書から分離してEPROMに焼くことができる。 これらの技術革新は、Forthコミュニティにおいて大きな影響を与え、より良い、より効率的なForthシステムやプログラムを生み出すことになるでしょう。 .new

1. The Kernel

The kernel in a Forth system is a collection of low level instructions which drive the computer and are used to construct other high level instructions. In cmFORTH, the kernel contains primarily simple NC4000 machine instructions. However, many of the commonly used Forth words do not have corresponding single word NC4000 instructions and they will have to be synthesized from the primitive NC4000 instructions.

1.1. The primitive Forth Words

Primitive stack operators are the machine instructions of the NC4000 chip. However, each machine instruction must have two versions, one to be executed by the text interpreter and the other for the compiler. The executable version must be a colon defintion with names so that they can be found by the text interpreter. They can be compiled into other colon definitions, but the resulting code will be terribly inefficient because of the overhead in nesting and unnesting. The other version used by the compiler is smart. It generates optimized machine code whenever feasible, taking advantage of the unique property of NC4000 in combining many Forth words into a single machine instructions.

The following primitive Forth words are redefined so that they can be executed by the text interpreter.

Forthシステムにおけるカーネルは、コンピュータを駆動し、他の高レベル命令を構築するために使用される低レベル命令のコレクションです。 cmFORTHでは、カーネルは主に単純なNC4000マシン命令を含んでいます。 しかし、よく使われるForthの単語の多くは、対応する一語のNC4000命令を持たないので、原始的なNC4000命令から合成する必要があります。

1.1. 原始的なForthの単語

原始スタック演算子はNC4000チップの機械命令である。 ただし、各機械命令には、テキストインタプリタが実行するバージョンと、コンパイラが実行するバージョンの2つが必要です。 実行可能なバージョンは、テキストインタプリタによって見つけられるように、名前を持つコロン定義でなければなりません。 他のコロン定義にコンパイルすることもできますが、ネストやアンネストのオーバーヘッドが発生するため、出来上がるコードはひどく非効率的なものになります。 コンパイラが使用するもう1つのバージョンはスマートです。 これは、多くのForth語を1つの機械命令にまとめるというNC4000のユニークな特性を利用して、実現可能な限り最適化された機械コードを生成します。

以下の原始的なForth語は、テキストインタプリタによって実行できるように再定義されています。

```
: SWAP   SWAP ;
: OVER   OVER ;
: DUP    DUP  ;
: DROP   DROP ;
: XOR    XOR  ;
: AND    AND  ;
: OR     OR   ;
: +      +    ;
: -      -    ;
: 0<     0<   ;
```

```
: NEGATE NEGATE ;
: @      @    ;
: !      !    ;
```

Many other commonly used Forth words cannot be reduced to single NC4000 instructions, so they have to be constructed with several NC4000 machine instructions.

その他、よく使われるForthの言葉の多くは、NC4000の単一命令に還元できないため、複数のNC4000マシン命令で構成する必要がある。

```
: ROT                       ( n1 n2 n3 -- n2 n3 n1 )
       >R SWAP              Exchange n1 and n2.
       R> SWAP              Exchange n1 and n3.
       ;

: 0=                        ( n -- f )
       IF 0 EXIT THEN       Return false if not 0.
                            EXIT is cheaper and faster.
       -1                   Can be obtained from a register.
       ;

: NOT                       ( n -- f )
       0=                   Logic NOT, not one's complement.
       ;

: <                         ( n1 n2 -- f )
       - 0<
       ;

: >                         ( n1 n2 -- f )
       SWAP-                A NC4000 primitive instruction.
       ;

: =                         ( n1 n2 -- f )
       XOR 0=
       ;

: U<                        ( u1 u2 -- f )
       - 2/                 Get the carry of subtraction.
       0<                   Return proper flag.
       ;

: ?DUP                      ( n -- n n | 0 )
       DUP
       IF DUP EXIT THEN     EXIT is faster.
       ;

: WITHIN                    ( n low high -- f )
       OVER - >R            high - low
       -                    n - low
       R> U<                In range?
```

```
        ;

: ABS                       ( n -- u )
        DUP
        0<                  Negative?
        IF NEGATE EXIT THEN Invert negative number.
        ;

: MAX                       ( n1 n2 -- n1 | n2 )
        OVER OVER -         n1 - n2
        0<                  Compare.
        IF SWAP-DROP EXIT   n1 < n2, drop n1.
        THEN DROP           Otherwise, drop n2.
        ;

: MIN                       ( n1 n2 -- n1 | n2 )
        OVER OVER -         n1 - n2
        0<
        IF DROP EXIT        n1 > n2, drop n2.
        THEN SWAP-DROP      Otherwise, drop n1.
        ;
```

The funny IF-BEGIN... UNTIL-THEN structure in Screen 11, connecting the two definitions MAX and MIN, in effect achieves the above function with a net saving of four instructions. You can do it because cmFORTH does not have compiler security and protection. Not recommended for general programming practice.

画面11の面白いIF-BEGIN... 画面11のUNTIL-THEN構造で、MAXとMINの2つの定義をつなぐことで、実質的には4命令の正味の節約で上記の機能を実現することができます。 cmFORTHにはコンパイラのセキュリティやプロテクションがないのでできることです。 一般的なプログラミングの実践にはお勧めできません。

```
: 2DUP                      ( d -- d d )
        OVER OVER
        ;

: 2DROP                     ( d -- )
        DROP DROP
        ;
```

1.2. Memory Accessing Words

```
: +!                        ( n a -- )
        0 @+                Fetch from a, while keeping a
                            on the stack.
        >R                  Save a.
        +                   Add n to contents of a.
        R> !                Store the sum back into a.
        ;
```

```
: 2C@+                      ( a -- a+1 l h )
       1 @+                 Fetch a cell and increment a.
       SWAP DUP             Get the contents just fetched.
       127 AND              Isolated the low byte.
       SWAP 6 TIMES 2/      Right justify the high byte.
       ;

: 2/MOD                     ( n -- rem quot )
                            Equivalent to   2 /MOD  but faster.
                            Needed to convert byte address
                            to cell address.
       DUP 1 AND            Get the remainder.
       SWAP 2/              Shift left to get quotient.
       ;

: C!                        ( b a -- )
                            a is byte address, which has to
                            be converted to cell address.
       2/MOD DUP >R         Save cell address.
       @                    Cell contents.
       SWAP                 Byte offset.
       IF -256 AND          Offset=1.  Mask off lower byte.
       ELSE 255 AND         Offset=0.  Mask off higher byte.
         SWAP               Get the byte b.
         6 TIMES 2*         Shift left by 8 bits.
       THEN
       XOR                  Combine two bytes.
       R> !                 Put the cell back.
       ;

: C@                        ( a -- b )
       2/MOD                Get the cell address.
       @                    Contents of the cell.
       SWAP 1 -             Offset=1 ?
       IF 6 TIMES 2/ THEN   Yes.  Shift right by 8 bits.
       255 AND              Mask off the high byte.
       ;

: 2@                        ( a -- d )
       1 @+                 Get the most significant cell.
       @                    Get the least significant cell.
       SWAP                 Put them in correct order.
       ;

: 2!                        ( d a -- )
       1 !+                 Store the most significant cell.
       !                    Store the next cell.
       ;
```

NC4000 is a 16 bit machine and it can access memory only by 16 bit cells. Two bytes are packed into one cell, with the first byte in the higher half (MSB) of the cell. The byte address is twice that of the cell address, with the least significant bit as the byte offset in a cell. To access one byte in the memory, one has to convert the

byte address to a cell address by 2/MOD and use the quotient as an offset to find the requested byte. It takes lots of extra work to do byte addressing. Avoid it at all cost.

If you really have to get bytes from the memory, the right word to use is 2C@+ which fetches one cell from the memory and returns both bytes on the stack. The address is incremented by 1 and preserved as the third element on the stack so you can fetch the next two bytes. It is faster than C@ and much more powerful.

NC4000は16ビットマシンであり、16ビットセルによってのみメモリにアクセスすることができます。 1セルに2バイトが詰め込まれ、1バイト目はセルの上位半分(MSB)にある。 バイトアドレスはセルアドレスの2倍で、最下位ビットがセル内のバイトオフセットとなる。 メモリ内の1バイトにアクセスするには、バイトアドレスを2/MODでセルアドレスに変換し、その商をオフセットとして使用して要求されたバイトを見つける必要があります。 バイトアドレッシングを行うには、多くの余分な作業が必要です。 何としても避けてください。

もし本当にメモリからバイトを取得する必要がある場合、使用する正しい単語は2C@+で、メモリから1つのセルをフェッチし、スタックに両方のバイトを返します。 アドレスは1つインクリメントされ、スタックの3番目の要素として保存されるので、次の2バイトをフェッチすることができます。 C@より速く、より強力です。

```
: MOVE                      ( a1 a2 n -- )
                            a1 is the starting address of the
                            source, and a2 is the end address
                            of destination string.  Be careful!
       >R                   Save cell count.
       MD I!                Save a2 in MD register.
       I TIMES 1 @+         Fetch n cells to the data stack.
       MD I@!               Retrieve a2.
       R>                   Retrieve cell count.
       TIMES 1 !-           Pop cells to destination in
                            reverse order.
       DROP                 Discard last address.
       ;

: FILL                      ( a # n -- )
                            Fill a memory range with cell value n.
       SWAP 1 - >R          Push n-1 on return stack as count.
       SWAP                 ( n a -- )
       BEGIN
         OVER SWAP          ( n n a -- )
         1 !+               Non-destructive store with a incremented.
       NEXT
       2DROP                Clear stack.
       ;

: ERASE                     ( a # -- )
                            Zero a range of cells.
       0 FILL
       ;
```

1.3. Multiply and Divide

NC4000 does not have single instruction multiply or divide, which requires a lot of gates to implement. What is provided are multiply steps, divide steps, and square-root steps, which can be used repetitively to achieve the desired results. Problems in processing the carry bit in the prototype chip cause some restrictions in multiplication. The software fixes to perform correctly the proper function are not implemented. You have to work around these bugs. An example is included in Chapter IV.

NC4000は単命令の乗算、除算を持たないので、実装に多くのゲートを必要とする。 そこで、乗算、除算、平方根の各ステップを用意し、これらを繰り返し使用することで、目的の結果を得ることができる。 試作チップではキャリービットの処理に問題があり、乗算に制約が生じることがある。 本来の機能を正しく実行するためのソフトウェアの修正が実装されていない。 このバグを回避する必要があります。 例題は第IV章に含まれています。

```
OCTAL

: U*+                       ( u1 r u2 -- d )
                            Unsigned integers u1 and u2 are
                            multiplied and added to r.  The
                            product is an unsigned double
                            integer on the stack.
                            Warning: u2 must be even!
       MD I!                Store u2 in MD register.
       16 TIMES *'          Repeat multiply step instruction
                            *' 16 times and the product is
                            left on the stack.
       ;

: M/MOD                     ( ud u -- q r )
                            Unsigned double integer ud is divided
                            by unsigned integer u.  Both quotient
                            and remainder are left on the
                            stack.  Note the order of q and
                            r is not standard.
       MD I!                Store u in MD register.
       D2*                  Left shift d by 1 bit so that
                            it is always even.
       15 TIMES /'          Repeat divide step /' 15 times.
       /''                  Last divide step.
       ;

: -M/MOD                    ( d u -- q r )
                            Double integer d is divided by
                            unsigned integer u.
       OVER 0<              Is d negative?
       IF
         DUP >R             Save u.
         +                  Add u to the higher half of d
       THEN                 for floored division.
       M/MOD                Do the divide now.
       ;

: M/                        ( d u -- q )
```

```
                              Mixed mode divide.
        -M/MOD DROP           Discard remainder.
        ;

: M*+                         ( u1 0 u2 -- d )
                              Unsigned multiply.
        MD I!                 Copy u2 to MD register.
        13 TIMES *'           Repeat multiply step.
        *-                    Last signed multiply step.
        ;

: VNEGATE                     ( n1 n2 -- n3 n4 )
                              Negate top two integers on the
                              stack.
        NEGATE                Negate top integer.
        SWAP NEGATE SWAP      Negate next integer.
        ;

: M*                          ( n1 n2 -- d )
                              Mixed mode multiplication of two
                              signed integers.
        DUP 0<                Is n2 negative?
        IF VNEGATE THEN       If so, negate both integers.
        0 SWAP                initialize accumulator.
        M*+                   Do the multiplication.
        ;

: /MOD                        ( u1 u2 -- r q )
                              Divide unsigned integers and return
                              both remainder and quotient.
        0 SWAP                Insert 0, making dividend a double
                              integer.
        M/MOD                 Do the mixed mode divide.
        SWAP                  Correct the order of results.
        ;

: MOD                         ( u1 u2 -- r )
                              Find remainder of unsigned integer
                              division.
        /MOD                  Do the generalized divide.
        DROP                  Discard quotient.
        ;

: */MOD                       ( u1 u2 u3 -- r q )
                              Multiply u1 and u2.  Divide the
                              double integer product by u3.
                              Return both remainder and quotient.
        >R                    Save divisor u3.
        0 SWAP U*+            Multiply u1 and u2.
        R> M/MOD              Divide by u3.
        SWAP                  Correct the order of r and q.
        ;

: */                          ( n1 n2 u -- r )
```

```
                              Ratio of n1*n2/u.
        >R                    Save u.
        M*                    Signed multiply of n1 and n2.
        R> M/                 Divide by u.
        ;

: *                           ( n1 n2 -- r )
                              Signed multiply.
        0 SWAP U*+            Signed multiply.
        DROP                  Discard high order cell.
        ;

: /                           ( n u -- q )
                              Divide by unsigned integer.
        >R                    Save divisor u.
        DUP 0<                Extend the sign of n.
        R> M/                 Divide.
        ;
.new
```

2. System Variables

System variables contain vital information needed by the Forth system to function. Most of them are pointers to various areas in the Forth system, such as the top of the dictionary, the disk buffers, the terminal input buffer, the vocabulary threads, etc. System variables in this implementation are kept at the bottom of RAM space, starting from location 16. Thus the first 16 system variables are in the so called local memory, which can be accessed by single cell instructions. These are the most frequently used system variables. Less frequently used variables are kept above location 35.

Following is the list of system variables defined in this implementation, their memory locations, their initial values if initialized, and their function.

システム変数には、Forthシステムが機能するために必要な重要な情報が含まれています。 そのほとんどは、辞書の先頭、ディスクバッファ、端末入力バッファ、語彙スレッドなど、Forthシステム内のさまざまな領域へのポインタである。 この実装におけるシステム変数は、RAM空間の最下部に位置する16番から保持されます。 したがって、最初の16個のシステム変数は、いわゆるローカルメモリにあり、シングルセル命令でアクセスすることができる。 これらは、最も頻繁に使用されるシステム変数です。 使用頻度の低い変数は、ロケーション35より上に保持されます。

以下に、この実装で定義されたシステム変数のリストとそのメモリ位置、初期化された場合の初期値、およびその機能を示します。

```
PREV                    Memory 16, not initialized.
                             Pointer to the most recently
                             referenced disk buffer.

OLDEST              Memory 17, not initialized.
                             Pointer to the oldest loaded
                             disk buffer.
```

```
BUFFERS             Memory 18 and 19, not initialized.
                           Storing block numbers of blocks
                           in the disk buffer.

NB                       A constant of value 1.
                           Number of disk buffers less 1.

BASE                   Memory 20, initialized to 10.
                           Number base for numeric I/O
                           conversion.

CNT                     Memory 21, not initialized.
                           Count of characters received from
                           the terminal device.

>IN                      Memory 22, not initialized.
                           Pointer to the input stream of
                           characters.  Used by WORD to parse
                           strings.

BLK                      Memory 23, initialized to 0.
                           Contains the block number under
                           interpretation.

?CODE                  Memory 24, initialized to 0.
                           Storing the address of the machine
                           code most recently compiled.
                           Used by the optimizing compiler
                           to construction multi-function
                           instructions.

dA                       Memory 25, initialized to 0.
                           Memory address offset to be subtracted
                           from the current address so that the
                           dictionary compiled can be relocated
                           to another part of memory.

MSG                    Memory 26, initialized to the
                           end of the system variable area.
                           Pointer to the terminal input
                           buffer.

CURSOR              Memory 27, initialized to 0.
                           Pointer to the memory location where
                           the last input character is stored.

WIDTH                Memory 28, initialized to 2.
                           Cells in the name field of an
                           entry in the dictionary.

OFFSET               Memory 29, initialized to 0.
                           Block number which became the
                           logic 0 block during disk access.
```

```
H                         Memory 30, initialized to 64 cells
                             after terminal input buffer MSG.
                             Pointer to the top of the current
                             dictionary.

C/B                       Memory 31, initialized to 417.
                             Machine cycles equivalent to the
                             width of a bit riding the RS-232
                             terminal port.

Interrupt Vector    Memory 32 to 33, not initialized.
                             Machine instructions are stored
                             here to handle interrupt requests.

Thread Table        Memory 34 and 35, pointers to the
                             ends of 2 threads in the dictionary.
                             The dictionary and vocabularies
                             are hashed into 2 threads.  The
                             name field addresses at the end
                             of each thread are stored in this
                             table for dictionary searching.

CONTEXT            Memory 36, initialized to 1.
                             Hash code of the context
                             vocabulary.
```

3. Terminal Input and Output

The terminal input and output in the RS-232 format is implemented through software via two I/O pins in the X-port, X0 as serial output and X4 as serial input. With the clock running at 4 MHz, the time interval representing one bit at 9600 baud is about 417 cycles, as specified by the system variable C/B. The primitive to send an ASCII character to the terminal is EMIT and that to receive a character from terminal is KEY. Line based I/O words TYPE and EXPECT, and all other terminal I/O words are derived from them.

3.1. Primitive Input and Output Words

```
: EMIT                       ( c -- )
                             Send a character to the terminal
                             via X0.
       30 13 I!              Mask X-port to allow X0 to be
                             output and other bits be input.
       2* 511 XOR            Make room for the start bit and
                                     invert polarity of bits.
       9 FOR                 Send out 8 data bits with one
                             start bit and one stop bit.
         12 I!               Send out one bit.
         2/                  Shift out next bit.
         C/B @ 11 - CYCLES   Wait for one bit period.
       NEXT                  Continue for the entire bit pattern.
       DROP                  Discard the rest of the character.
```

```
        ;

: RX                        ( -- n )
                            Get one bit from X4 pin.
        12 I@               Read the X-port.
        16 AND              Save only the X4 pin input.
        ;

: KEY                       ( -- c )
                            Read one ASCII character from
                            X4 pin.
        0                   Starting character pattern.
        BEGIN               Wait for the start bit.
          RX                Read the input line.
          16 XOR            Exit only when a start bit (low)
        UNTIL               is detected.
        C/B @               417 cycles per bit.
        DUP 2/ +            Wait 1.5 bit to the center of
                            the first data bit.
        7 FOR               Read 8 bits.
          14 - CYCLES       Delay till the center of bit period.
          2/                Ready the character pattern for
                            the next bit.
          RX                Read one bit.
          2* 2* 2*          Justify the bit position.
          OR                Put the bit into the character
                            pattern.
          C/B @             Delay for next bit.
        NEXT                Repeat until all eight bits are
                            assembled in the character pattern.
        BEGIN RX UNTIL      Now wait until the stop bit is
                            transmitted.
        DROP                Discard the last C/B cycle number.
        ;
```

3.2. The Line Input and Output Words

```
: TYPE                      ( a1 -- a2 )
                            Output a stored string to the
                            terminal.  The first character
                            in the string must be a count
                            byte.  This is different from
                            the standard TYPE which takes
                            an address and a count as arguments.
                            a1 is the starting cell address
                            and a2 is the address of the cell
                            following the string.
        2*                  Change a1 to a byte address.
        DUP C@ 1 -          Get the count byte.
        FOR                 Scan the string.
          1 +               Next character address.
```

```
          DUP C@              Get the character.
          EMIT                Send it out.
        NEXT
        2/                    Cell address after the string.
        ;

: EXPECT                      ( a n -- )
                              Accept n characters and put them
                              in the memory starting at a.
                              Each character is put in a cell
                              with high byte padded with 40H.
        SWAP CURSOR !         Store address a in CURSOR.
        1 - DUP FOR           Repeat for n characters.
          KEY                 Get one character.
          DUP 8 XOR           Is it a backspace?
          IF                  No.  Not backspace.
            DUP D XOR         Is it a carriage return (CR)?
            IF                Not CR.
              DUP 4000 +      Pad it with a @.
              CURSOR @ 1 !+   Store it in the assigned memory.
              CURSOR !        Refresh CURSOR for next character.
              EMIT            Echo the character to terminal.
            ELSE              If it is CR.
              SPACE           Output a space instead.
              DROP            Discard the CR character.
              R>              Get the current index.
              -               Number of character received so
                              far.
              CNT !           Store it in the character count
                              variable CNT.
              EXIT            CR end of line exit.
            THEN
          ELSE                Yes.  It is backspace.
            DROP              Discard the backspace character.
            DUP I XOR         If the backspace is the first
                              character in the input stream,
            [ OVER ] UNTIL    return to the beginning of the
                              FOR-NEXT loop immediately.
                              [ OVER ] UNTIL compiles a
                              conditional branch to the
                              address left on stack by FOR.
            CURSOR @ 1 -      Get the cursor address again,
            CURSOR !          and decrement it.
            R> 2 + >R         Add 2 to loop index to back
                              up the character pointer by 1.
            8 EMIT            Echo the backspace code.
          THEN
        NEXT
        1 + CNT !             Increment character count.
        ;
```

3.3. Other Terminal I/O Words

```
: CR                        ( -- )
       13 EMIT              Carriage return.
       10 EMIT              Line feed.
       ;

: SPACE                     ( n -- )
       32 EMIT
       ;

: SPACES                    ( n -- )
       0 MAX                Protection against negative
                            count number.
       ?DUP IF              More than one character?
         1 - FOR            Yes.
           SPACE            Send them out.
         NEXT
       THEN
       ;

: HERE                      ( -- n )
       H @                  Top of dictionary or the WORD
                            buffer.
       ;
.new
```

4. The Number Conversion Words

Number is the most important entity used in a computer. It has to be entered in ASCII digit strings and converted into the binary representation for the computer to operate on. The result thus generated will have to be converted back to human readable digit strings so that the user can make some sense of it. Forth has the best user interface as far as numbers are concerned. Input numbers are converted to binary form and pushed onto the data stack. Output numbers can be displayed in several formats depending upon the user's need. The user can select any reasonable number base for the conversion between the ASCII form and the binary form.

4.1. Convert Digits to Binary Number

Strings of digits or numbers are one of the two basic syntactic elements in the Forth language. The numbers typed in by the user must be converted to 16 bit binary numbers and pushed onto the data stack. The conversion process is controlled by the variable BASE which specifies the number base to be used in the conversion process.

```
HEX                         Change to hexadecimal base because
                            we will use ASCII code values.

: -DIGIT                    ( c -- n )
                            Convert one ASCII character c
                            to its binary value n.  Abort
                            it the character is not within
```

```
                              the range specified by BASE.
        DUP 39 >              Is c greater than 9?
        IF                    Yes.
          DUP 40 >            Is it also greater than @?
          7 AND -             If so, subtract 7 to take care
                              of the gap between 9 and A.
        THEN
        30 -                  Take off the offset to 0.
        DUP BASE @ U<         Is the result less than base?
        IF EXIT THEN          If so, the conversion is successful.
                              Return with the value.
        2DROP DROP            Otherwise, conversion error.
                              Clear the stack.
        ABORT" ?"             Abort with a message ?.
        DROP ; RECOVER        Forcing the compilation of a DROP
                              and ; instruction and eliminate
                              this instruction to save 1 cell
                              of memory.  It has to be done
                              this way because ABORT" ?" compiles
                              a string literal which is inappropriate
                              to hang a return bit.

: 10*+                        ( u1 c -- u2 )
                              Convert character c to its value.
                              Multiply it by the base value hidden
                              in MD and accumulate the product
                              into u1.
        -DIGIT                Convert c to its binary value.
        0E TIMES *'           Repeat multiply step 16 times to
                              obtain the product.
        DROP                  Discard the higher half of the
                              double integer product.
        ;

: NUMBER                      ( a -- n )
                              Convert a digit string at a to
                              a 16 bit integer.
        BASE @ MD I!          Store base value in MD register.
        2C@+                  Get first two characters of the
                              string.
        OVER 2D = DUP >R      If the first character is a minus
                              sign, save a true flag on return
                              stack.
        IF                    Yes, it is a minus sign.
          SWAP-DROP           Discard the minus character.
          0                   Initialize accumulator for conversion.
        ELSE                  Not a minus sign.
          SWAP -DIGIT         Convert the first character and
                              use it as the accumulator.
        THEN SWAP             ( next-addr accumulator count )
        1 - ?DUP              More than one character in the
                              string?
        IF                    Yes.
          1 - 2/ FOR          Run down the digit string.
```

```
            SWAP 2C@+          Get next two characters.
            SWAP >R >R         Get the character out of the way.
            SWAP               Swap the next address with the
                               accumulator.
            R> 10*+            Convert one character and add
                               its value to the accumulator.
            R> 20 XOR          Is the next character a blank?
            IF 10*+            No.  Convert it.
            ELSE DROP THEN     Yes.  Drop the blank.
          NEXT                 Continue for entire string.
        THEN
        SWAP-DROP              Discard the address.
        R>                     Retrieve the sign flag.
        IF NEGATE THEN         Negate the number if it has a
                               leading - sign.
        ;
```

4.2. Convert Binary Number to ASCII String

This conversion process is different from those we know well in other Forth systems. The converted digits are piled up on the stack instead of stored in an output buffer. This method is much more efficient, as it eliminates the need for a buffer with pointer and management overhead.

```
: HOLD                         ( .. # n c -- .. # n )
                               The output string is piled up
                               on the data stack with a count
                               on top.  Above count #, the number
                               to be converted n and the character
                               c to be added to the string.
                               c is tucked beneath # and # is
                               incremented.
        SWAP >R                Save n.
        SWAP                   Tuck c under #.
        1 +                    Increment count #.
        R>                     Retrieve n.
        ;

: DIGIT                        ( n -- c )
                               Convert a number n to its equivalent
                               ASCII code.
        DUP 9 >                Is it greater than 9?
        7 AND +                If so, add 7 to jump to A.
        48 +                   Add offset of 0 in the ASCII scale.
        ;

: <#                           ( n -- # n )
                               Prepare a number to start the
                               conversion process.
        -1                     Initial value of the string length.
        SWAP                   Tuck the count # under n.
        ;
```

```
: #                         ( .. # n -- .. #' n' )
                            Convert one digit from n and add
                            the converted digit to the output
                            string.
       BASE @ /MOD          Divide n by the base.
       SWAP DIGIT           Convert the remainder to an ASCII
                            character.
       HOLD                 Add the converted character to
                            the output string.
       ;

: #S                        ( .. # n -- .. #' 0 )
                            Convert the number n until it
                            is reduced to 0, or completely
                            converted.
       BEGIN
         #                  Convert one digit.
         DUP 0=             End?
       UNTIL
       ;

: #>                        ( .. # n -- )
                            Output the converted string to
                            the terminal.
       DROP                 n is usually 0.  It is not needed
                            in any case.
       FOR EMIT NEXT        Use the character count # to print
                            that many characters to the terminal.
       ;

: SIGN                      ( .. # n -- .. #' )
                            If n is negative, append a - sign
                            to the end of the output string.
       0<                   Is n negative?
       IF 45 HOLD THEN      If so, append - sign.
       ;

: (.)                       ( n -- .. # )
                            Convert the number n to a ASCII
                            string on stack.
       DUP >R               Save a copy of n for sign.
       ABS                  Take the absolute value of n.
       <# #S                Convert the absolute value to
                            string.
       R> SIGN              Append the sign of n.
       ;

: .                         ( n -- )
                            Free format display of the number
                            on top of the stack.
       (.)                  Convert n to a string.
       #>                   Print the string.
       SPACE                Append a space to separate consecutive
```

```
                              numbers.
        ;

: ?                           ( a -- )
                              Display the contents of a memory
                              cell.
        @ .                   Get the number and display it.
        ;

: U.R                         ( u n -- )
                              Display an unsigned integer u
                              in a field of n columns, right
                              justified.  Formatted output.
        >R                    Save the column number n.
        <# #S                 Convert u to a string.
        OVER                  Copy character count to top.
        R> SWAP-              Subtract it from the column width.
        1 - SPACES            First pad the left side with enough
                              spaces.
        #>                    Finally print the number string,
                              right justified.
        ;

: U.                          ( u -- )
                              Display an unsigned integer in
                              free format.
        0 U.R                 Display the integer using 0 column
                              field specification.  The result
                              is that the string will be display
                              from the current character position.
        SPACE                 Followed by a space.
        ;
```

4.3. Memory Dump

This memory dump word was designed for Chuck Moore's peculiar CRT display, which has a single line display window.

```
: DUMP                        ( a -- a+8 )
                              Display 8 consecutive cells following
                              the cell at a.  a+8 is returned on the
                              stack so another DUMP can be issued.
        CR                    New line.
        DUP 5 U.R SPACE       Display first the address a.
        7 FOR                 Run down 8 cells.
          1 @+                Fetch one cell and increment a.
          SWAP 7 U.R          Display the contents.
        NEXT
        SPACE                 Add one space at the end of line.
        ;
```

4.4. Message Output

In an interactive programming environment, it is important that the system sends timely messages to the terminal to show the user its current status and any error condition. Messages to be send to the terminal must be compiled into definitions using special string literal words like ." and ABORT" , etc. These string literal words have unique behavior during compilation and during execution. Because these words involve compiler functions, words used to define them seem to be out of place as they are defined much later than the text interpreter. From among these special words, the ones relevent to the construction of message output words are excerpted here. Some of these words will be explained later in more detail.

```
: COMPILE                   ( -- )
                            Compile the address following
                            this word to the top of the dictionary.
       R>                   Retrieve the address of the next
                            word from the return stack.
       7FFF AND             Mask off the most significant
                            bit, which is the carry bit.
       1 @+                 Fetch next word.
       >R                   Put the address of the second
                            word after COMPILE back on the
                            return stack.
       ,A                   Compile the address to the
                            dictionary.
       ;

: abort"                    ( -- n 0 )
                            Run time routine for ABORT".
                            Print the following message and
                            reinitialize the system.
       H @  TYPE SPACE      Display the name of word currently
                            been executed.
       R> 7FFF AND          Address of the compiled text.
       TYPE                 Display the message text.
       2DROP                Clean the garbage left by TYPE.
       BLK @                Leave the block number on stack
                            as a debugging aid.
       QUIT                 Return to the text interpreter.
       ;

: ABORT"                    ( -- )
                            Abort the word currently been
                            executed and return to the text
                            interpreter after the following
                            message is displayed.
       COMPILE abort"       Compile the runtime routine.
       4022 STRING          Compile the following message
                            up to " as a string literal.
       ;

: dot"                      ( -- )
                            Run time routine of ." , which
```

```
                            displays the message immediately
                            following until " .
        R> 7FFF AND         Address of the compiled message
                            text.
        TYPE                Display the message and leave
                            the address after the message.
        >R                  Push that address back on the
                            return stack to continue the execution
                            process.
        ;

: ."                        ( -- )
                            Display the following message
                            at run time.
        COMPILE dot"        Compile the address of dot" .
        4022 STRING         Compile the following text up
                            to " as a string literal.
        ;
```

5. Serial Disk

Here we assume that there is only a RS-232 interface to the outside world. As with any other computer language for serious programming activity, one or more disk drives are necessary to store source code and data. A serial disk is thus designed to make the maximum utilization of the available serial communication line. The serial disk requires a host computer at the other end of the RS-232 line to act as the terminal and disk server for this Forth system. Whenever a disk block is requested, the data will be transferred into the Forth system through the serial link. When an updated block is flushed back to the disk, the data is also sent through the serial link.

5.1. Disk Buffer Manager

Two disk buffers are maintained in the cmFORTH system. Each buffer is 1024 cells long. The first buffer starts at memory address 800H and the second buffer is at C00H under the ROM memory. Two cells in BUFFERS array contain the block numbers associated with the data stored in the two buffers. The pointer PREV, points to the disk buffer which was referenced most recently. OLDEST points to the disk buffer least used.

In this system, two disk buffers are assigned and numbered as 0 or 1. Two entries in the BUFFERS array are used to store block numbers corresponding to the contents of the two buffers. Two variables PREV and OLDEST determined which of the two buffers is the most recently accessed. The manager always looks at the PREV block when a block is requested. If the block is not in the PREV buffer, it will exchange PREV with OLDEST and look at the PREV block again. If the requested block is in one of the two buffers, that buffer will certainly become the PREV buffer and no disk access will be necessary. If the requested block is not in these buffers, the manager will assign the PREV buffer to the new block; and the old data in this buffer which must be the least recently referenced block will be flushed to the disk or discarded.

This technique is often referred to as the Ping-Pong buffers. The two buffers are used in the most efficient fashion.

```
: ADDRESS                   ( n -- a )
                            Given the disk buffer number,
                            return the starting address of
                            that buffer.
       2 +                  Offset of 2048 cells.
       8 TIMES 2*           Multiplied by 1024 to get the
                            buffer address.
       ;

: ABSENT                    ( n -- n | a )
                            Search through the disk buffers
                            to see if block n is already in
                            one of the buffers.  If found,
                            return the address of the buffer
                            and skip the next word.  Otherwise,
                            return with n on the stack.
       NB FOR               Scan through the disk buffers.
         DUP                Copy n, the requested block number.
         I BUFFERS @        Get one block number stored in
                            BUFFERS.
         XOR 2*             Are the 15-bit block numbers the
                            same?
       WHILE NEXT           If not the same, compare the next
                            block number in BUFFERS.
       EXIT THEN            None of the buffers contains the requested
                            block, return as if nothing had happened.
       R> PREV N!           At this point, the request block is
                            found in one of the buffers.
                            Store the buffer number in PREV,
                            and make it the most recently accessed
                            block.
       R>DROP               Discard the return address on

                            top of the return stack, thus
                            exiting the word containing ABSENT.
       SWAP-DROP            Discard the block number.
       ADDRESS              Return with the buffer address.
       ;

: UPDATED                   ( -- a n )
                            Exchange PREV and OLDEST buffers
                            and return the address and the
                            block number of the least recently
                            accessed buffer.  If the block
                            is not updated, skip rest of the
                            words following UPDATED.
       OLDEST @             Pointer to the buffer least recently
                            used.
       BEGIN
         1 + NB AND         Map to one of the two buffers
                            allocated in this system.
         DUP                Save a copy.
         PREV @ XOR         Is it the same as the one stored
```

```
                              in PREV?
        UNTIL                 Exit if they are different.
        OLDEST N! PREV N!     Exchange contents of OLDEST and
                              PREV, thus making OLDEST the most
                              recently accessed disk buffer.
        DUP ADDRESS           Find the address of the buffer.
        SWAP BUFFERS          Obtain the right pointer to the
                              BUFFERS array.
        DUP @                 Get the block number stored in
                              BUFFERS.
        8192 ROT !            Store 2000H in this entry of BUFFERS.
        DUP 0< NOT            If the buffers is not updated,
        IF R>DROP DROP THEN   skip rest of the words following
                              UPDATED by thrashing the return
                              address on the top of return stack.
                              It is a very fast and implicit
                              EXIT.
        ;

: UPDATE                      ( -- )
                              Set the MSB of the block number
                              in BUFFERS pointed to by PREV.
        PREV @ BUFFERS        Address of the PREV block number.
        0 @+                  Fetch block number while still saving
                              the address of PREV.
        SWAP 32768 OR         Set bit 15.
        SWAP !                Put it back.
        ;

: ESTABLISH                   ( n a -- a )
                              Mark the oldest buffer the PREV
                              buffer and identifies it with
                              block n.  Return the buffer address
                              a.
        SWAP                  Get n to the top.
        OLDEST @ PREV N!      Make oldest the newest.
        BUFFERS !             Store block number n into the
                              BUFFERS array.
        ;

: IDENTIFY                    ( n a -- a )
                              Make block n the PREV block, as the
                              one most recently referenced.
        SWAP                  Get n .
        PREV @ BUFFERS !      Store n into the PREV entry in
                              the BUFFERS array.
        ;
```

5.2. Disk Read and Write

The serial disk is implemented using a very simple protocol. A disk read/write request is initiated by sending a NUL byte to the host at the other end of the RS-232 line, followed by two more bytes identifying the disk

block requested. High byte of the block number is send first. If the most significant bit in the high byte is set to 1, it is a disk write command. 1024 bytes will then be transmitted to the host. Then it will wait for a key from the keyboard to confirm the termination of transmission. If the MSB in the high byte is reset, it is a read command and the host is expected to send 1024 bytes of the requested block.

```
: ##                        ( a n -- a a 1023 )
                            Transmit a read disk command to
                            host and prepare to receive 1024
                            bytes.
       0 EMIT               Disk accessing command.
       256 /MOD EMIT EMIT   Transmit the read block command.
       DUP 1023             Parameters needed to receive the
                            requested block.
       ;

: buffer                    ( n -- a )
                            Return the buffer address of block
                            n.  If the buffer had been updated,
                            flush its contents to the host.
       UPDATED              If block n is already in one of
                            the disk buffers, return the buffer
                            address, and return to caller
                            immediately without executing
                            the following words.
       ## FOR               Block n is not in the disk buffers.
                            Get the least used buffer and
                            flush its contents to host if
                            the buffer was updated.
         1 @+               Fetch one cell.
         EMIT               Transmit one byte.
       NEXT
       KEY                  Wait for user response as end
                            of transmission.
       2DROP                Clean up.
       ;

: BUFFER                    ( n -- a )
                            Obtain a disk buffer for block
                            n and return the buffer address
                            a.
       buffer               Do all the hard work to obtain
                            a disk buffer, including flushing
                            if necessary.
       ESTABLISH            Mark this disk buffer as the most
                            recently accessed.
       ;

: block                     ( n a -- n a )
                            Read 1024 bytes from the host
                            and put them in the buffer starting
                            at a.
       OVER ##              Transmit the read command.
```

```
        FOR                     Repeat 1024 times.
          KEY 16384 XOR         Get one byte and stuff high byte
                                with 40H.
          SWAP 1 !+             Store the cell into disk buffer.
        NEXT
        DROP
        ;

: BLOCK                         ( n -- a )
                                Read block n from the host if
                                it is not already in the buffer.
                                Return the buffer address.
        ABSENT                  If block n is not in one of the
                                buffers, do the following to read
                                it from the host.  Otherwise,
                                return the buffer address and
                                exit here.
        buffer                  Make room in the least used buffer,
                                and flush its contents if updated.
        block                   Read from host.
        ESTABLISH               Make the buffer most recently
                                accessed.
        ;

: FLUSH                         ( -- )
                                Write all updated buffer back
                                to disk-host.
        NB FOR                  Go through all disk buffers.
          8192 BUFFER           Request block 8192, the default
                                empty block code.
          DROP                  Discard the buffer address.
        NEXT
        ;

: EMPTY-BUFFERS                 ( -- )
                                Erase all buffer pointers to
                                make the disk manager think
                                the buffers are empty.
        PREV                    Address of the PREV variable.
        [ NB 3 + ] LITERAL      The array including PREV, OLDEST,
                                and BUFFERS.
        ERASE                   Erase all these pointers to fool
                                the disk manager.
        FLUSH                   Initialize the buffers.
        ;
```

6. The Text Interpreter

The text interpreter is the operating system of Forth and it is the user interface which allows a user to operate the computer interactively. What the text interpreter does is very simple. It accepts a line of commands from the terminal, parses out words in the command line and executes them in the order given. It only has to deal with two types of words, Forth commands which had been compiled into the dictionary and numbers as 16 bit

integers. If the interpreter finds a word in the dictionary, it will execute that word. If the word is not defined in the dictionary, text interpreter will try to convert it into an integer and push the integer on the stack. If it fails to convert the word into a number, the word is outside of the computer's vocabulary and it will abort the command line. It will then come back and ask the user to type another command line and the process continues on for ever.

The major functions performed by the text interpreter are receiving command lines, parsing words, dictionary searches, command execution, and number conversion. We have already discussed number conversion and user input functions. Here we shall discuss the rest of the functions and how they are tied together to form a complete operating system.

6.1. Parsing of Words

```
: LETTER                    ( a1 a2 n -- a3 a4 )
                            Copy n characters from a2 to a1.
                            Source strings are stored in cells
                            and destination strings are stored
                            in bytes.  Terminate the copying
                            when a delimiter is detected.
                            The delimiter is stored in register
                            SR.
       FOR                  Scan n characters.
         DUP @              Get one character from a2.
         SR I@ XOR          Is the character the same as the
                            one stored in SR, the delimiter?
       WHILE                Not equal.
         SWAP >R            Save a1.
         1 @+               Fetch character and increment
                            a2.
         SWAP 127 AND       Retain only the lower byte.
         I C!               Store the character in a1.
         R> 1 + SWAP        Increment a1.
       NEXT
       EXIT THEN            Exit if the string is completely
                            copied.
       R>                   A delimiter was encountered.
                            Retrieve the index count.
       NEGATE >IN +!        Move the interpreter pointer >IN
                            back that many characters.
       ;

: -LETTER                   ( a1 a2 # -- a3 a4 )
                            Scan characters stored in buffer
                            a2.  Ignore leading delimiters
                            by comparing with SR.  Then move
                            the string into buffer a1.  Again,
                            a1 and a3 are byte addresses and
                            a2 and a4 are cell addresses.
       ?DUP IF              n has to be greater than 0.
         1 - FOR            Repeat n times.
           1 @+             Read one cell.
```

```
            SWAP SR I@ XOR    Is the character same as the one
                              in SR?
          0= WHILE NEXT       Yes.  Skip it and continue on.
          EXIT THEN           If the character string is exhausted
                              without finding the character
                              in SR, exit here.
          1 -                 Backup a2 by one cell.
          R>                  Index of the do-loop when branched
                              out at WHILE.
          LETTER              Scan the rest of the string and
                              copy it into a1 buffer.
        THEN
        ;

: WORD                        ( n -- a )
                              Parse out the next word from the
                              input buffer, using n as the delimiter.
                              The parsed word is placed in the
                              buffer at address a as a packed,
                              count string.
        >R                    Save n, the delimiter.
        H @  2* 1 +           Byte address of the destination
                              string buffer.  Leave one byte
                              for the length of string.
        DUP                   Need two copies of this byte address.
        >IN @                 Character pointer of the parser.
        BLK @ IF              If BLK is not zero, we are processing
                              text in a disk buffer.
          BLK @ BLOCK         Get the disk block and the buffer
                              address.
          +                   Address of the character cell
                              currently being processed.
          1024                Maximum characters in the disk buffer.
        ELSE                  BLK is 0.  Input is from the terminal
                              input buffer.
          MSG @ +             Address of the character in buffer
                              to be interpreted.
          CNT @               Total number of characters received.
        THEN
        >IN @                 Interpreter pointer.
        OVER >IN !            Save the total character count
                              in >IN.
        -                     Remaining character count between
                              interpreter pointer and end of
                              input buffer.
        R> SR I!              Store the delimiter in SR register.
        -LETTER               Parse out the next word and copy
                              it into the word buffer above
                              HERE.
        DROP                  Discard the input buffer address.
        32 OVER C!            Append a space after the parsed
                              word.
        SWAP-                 Character count of the parsed
                              word.
```

```
        H @  2* C!          Store the count at the beginning
                            of the parsed word as a packed
                            count string.
        H @                 Return the address of the word
                            buffer.
        ;
```

6.2. Dictionary Search

With the NC4000 chip, the code field and the parameter fields in a regular Forth system must be merged into a single field, as the inner interpreters NEXT, NEST (DOCOL), and UNNEST (;S) are all taken care of by hardware. A word defined in this system thus consists of three fields: a link field, a name field, and the code/parameter field. The bit arrangements in the name field can be shown as follows:

```
        r0sn nnnn tccc cccc          r: remote bit
        0ccc cccc 0ccc cccc          s: smudge bit
                ...                  n: character count
                ...                  t: truncation bit
        0ccc cccc tccc cccc          c: ASCII character
```

The truncation bit in the last cell of the name field indicates to the text interpreter that a long name is truncated and name comparison must stop at that cell.

The dictionary contains two vocabularies, FORTH and COMPILER. The link field addresses of the last words in these vocabularies are stored in an array, immediately prior to the system variable CONTEXT. Each cell in this array contains a pointer, pointing to the link field of the last word defined in the corresponding vocabulary. Each link field contains a pointer pointing to the link field of the previously defined word. The other end of the thread is the first word in the linked chain, whose link field contains a zero as the end-of-link indicator. To locate a word in a vocabulary, the link field address is first obtained from the BUFFERS array in front of CONTEXT. Then the linked chain is followed to see if the parsed string can match a name in that vocabulary.

```
HEX

: SAME                      ( a1 a2 -- a3 a4 f | a1 t )
                            With a string address a1 and the
                            link field address of a word in
                            dictionary, compare the string
                            with the word name.  If the name
                            matches the string, return the
                            code and link field addresses
                            of the word with a false flag.
                            Otherwise, return the string address
                            a1, link field address of the
                            next word in the linked chain
                            and a true flag.
        OVER >R             Save a copy of a1 on return stack.
        DUP                 Copy a2.
        1                   Offset to the name field.
```

```
        BEGIN
          + 1 @+              Get one cell from the name field.
          SWAP                Get the contents to top of stack.
          R>                  Address of a cell in the string.
          1 @+                Fetch one cell from there.
          R>                  Replace the string address.
          - 2*                Compare contents of the two cells,
                              ignoring bit 15.
          DUP                 Need a dummy zero when looping
                              back, to be consumed by +.
        UNTIL                 Exit if the cells are not equal.
        R>DROP                Discard the string address on
                              the return stack.
        FEFF AND              Is bit 8 of the difference set?
                              This is the terminator bit (bit 7)
                              in the last cell of a word name,
                              shifted to here by 2* above.
        IF 0 AND EXIT THEN    Exit with a false flag.  The names
                              failed to match.
        SWAP 1 + @            Get the next cell in the string buffer.
        0<                    If it is a string delimiter,
        IF @ THEN             fetch the link field address.
        SWAP                  Adjust the code field address
                              of the found word.
        ;

: HASH                        ( n -- a )
                              Use the vocabulary code n to find the
                              pointer to the head of thread in
                              the thread table before CONTEXT.
        CONTEXT SWAP-
        ;

: -FIND                       ( a1 a2 n -- a1 t | a2 f )
                              With word address a1, link address
                              a2 and vocabulary mask n, search
                              the link for the word.  If found,
                              return the code and link field addresses
                              of the word and a false flag.
                              If not found, return the word
                              address and a true flag.
        HASH                  Find the head of the thread.
        BEGIN                 Start the dictionary search.
          @ DUP               Get the next link field address.
        WHILE                 If link field address is not zero,
                              continue the search.  Otherwise,
                              the end of link chain is reached.
          SAME                Compare the name field with the
                              word pattern.
        UNTIL                 Not same.  Continue with the next
                              word in the linked chain.
        0 EXIT THEN           Find a word.  Return its address
                              and a false flag.
        -1 XOR                End of the linked chain.  Return
```

```
                              with the flag.
        ;
```

7.3. Control Structures

The NC4000 has three branch instructions: unconditional branch, conditional branch and loop. A branch instruction takes a 12-bit argument to specify a branch address, within a 4K word memory page. These instructions are used to implement various control structures in high level Forth definitions.

The conditional branching structures are of the following two types:

```
        ...  IF ... ELSE ... THEN  ...
        ...  IF ... THEN ...
```

There are several types of indefinite loops which can be constructed very easily with the conditional and unconditional branch instructions. The ones this cmFORTH system supports are:

```
        ...  BEGIN ... UNTIL  ...
        ...  BEGIN ... AGAIN  ...
        ...  BEGIN ... WHILE  ... REPEAT  ...
        ...  BEGIN ... WHILE  ... UNTIL ... THEN  ...
```

WHILE can branch to REPEAT or to THEN. The latter construction allows additional freedom, in that there are two distinct paths after AGAIN.

Definite loops are constructed with FOR and NEXT:

```
        ...  FOR ... NEXT  ...
```

which is very similar to the DO-LOOP structure in conventional Forth we all love. However, FOR takes only one parameter which is decremented every time through NEXT. The loop will be terminated when this index is decremented to zero.

The FOR-NEXT definite loop can also make use of the WHILE-THEN conditional:

```
        ...  FOR ... WHILE ... NEXT ... ELSE ... THEN  ...
```

However, one will have to take care of the loop index on the return stack, when the loop is terminated through WHILE. WHILE does not modify the return stack.

```
OCTAL                       For instructions and addresses.
```

```
: OR,                       ( a n -- )
                            OR the address a into the instruction
                            n and compile the branch instruction.
       0 ?CODE !            Start a new machine instruction.
       SWAP 7777 AND        Keep only the lower 12 bits in
                            address a.
       OR                   Include truncated address into
                            the branch instruction n.
       ,                    Compile the branch instruction.
       ;

: begin                     ( -- a )
                            Mark the current dictionary pointer
                            as address to be branched to.
       H @                  Push the current dictionary pointer
                            on the data stack.
       0 ?CODE !            Initialize the optimizer.
       ;

: BEGIN                     ( -- a )
                            Starting pointer of a indefinite
                            loop.
       begin                Leave the current dictionary pointer
                            on stack for later resolution.
       ;                    Compiler directive must be executed
                            in a definition.

: UNTIL                     ( a -- )
                            Compile a conditional branch to
                            address a.
       110000               Conditional branch instruction.
       OR,                  Add address and compile it.
       ;

: AGAIN                     ( a -- )
                            Compile an unconditional branch
                            to address a.
       130000               Unconditional branch instruction.
       OR,                  Add address and compile.
       ;

: THEN                      ( a -- )
                            Resolve the branch address in
                            the branch instruction compiled
                            by IF or ELSE.
       begin                Get the address of the current
                            instruction, as pointed to by
                            the dictionary pointer.
       7777 AND             Keep only the 12 bit part.
       SWAP +!              Add it into the 12 bit address
                            field in the IF or ELSE instruction.
       ;

: IF                        ( -- a )
```

```
                              Compile a conditional branch instruction
                              now and leave its address on the
                              stack so that its address field
                              can be resolved by ELSE or THEN.
        begin                 Leave the address of the unconditional
                              instruction on the stack.
        110000 ,              Compile a conditional branch instruction
                              with an unresolved address field.
        ;

: ELSE                        ( a1 -- a2 )
                              Resolve the conditional branch
                              instruction at a1. Compile an
                              unconditional branch instruction
                              with a 0 address field.  Leave
                              its address on the stack as a2,
                              to be used by THEN to resolve.
        begin                 Address of the current unconditional
                              branch instruction.
        130000 ,              Compile an unresolved unconditional
                              branch instruction.
        SWAP                  Get a1 to top of the stack.
        [COMPILE] THEN        Invoke THEN to resolve the conditional
                              branch instruction left by IF.
        ;


: WHILE                       ( a1 -- a2 a1 )
                              Compile an unresolved conditional
                              branch instruction.  Leave its
                              address on the stack as a2 while
                              passing the address left by BEGIN.
        [COMPILE] IF          Invoke IF to compile a conditional
                              branch.
        SWAP                  Exchange a1 and a2 so that they
                              can be used by REPEAT|AGAIN and THEN
                              to resolve the branch addresses.
        ;

: REPEAT                      ( a1 a2 -- )
                              Resolve the BEGIN-WHILE-REPEAT
                              structure.
        [COMPILE] AGAIN       Compile an unconditional branch
                              back to BEGIN, using address a2.
        [COMPILE] THEN        Resolve the conditional branch
                              instruction compiled by WHILE.
        ;

: FOR                         ( -- a )
                              Start a definite loop.
        [COMPILE] >R          Compile a to-R instruction which
                              saves the loop count in the I
                              register.  Leave the address of
                              the next instruction on stack
```

```
                              for the later NEXT instruction.
        begin                 Leave address of the next instruction
                              for NEXT to branch to.
        ;

: NEXT                        ( a -- )
                              Compile a loop instruction and
                              use the address a on the stack
                              for the branch address.
        120000                Code for the loop instruction.
        OR,                   Resolve the backward jump address.
        ;
```

7.4. The NC4000 Assembler

Assembler!? Good grief!

Supposedly, the NC4000 chip will speak high level Forth and we shall all be freed from the tyranny of assembler and live happily forever. The truth is that we can program in Forth and NC4000 will run the program much faster than anything we had previously. However, if we wish to squeeze the most out of it, we still have to deal with it by bits and pieces at the machine code level.

The NC4000 machine instruction problems can be handled in two different ways: Map the NC4000 instruction set onto a regular Forth instruction set and solve the problem with the regular Forth programming technique; or find ways to squeeze as many functions into a single instruction as possible in order to save both machine cycles and memory space. Here we shall be concerned with single function NC4000 instructions. We will show how they can be defined and how they are used to allow us to program in the usual Forth style. In the section on the Optimizing Compiler, we will discuss how a program might be optimized by combining many functions into single instructions.

```
: uCODE                       ( n -- )
                              Define an NC4000 machine instruction
                              and give it a name.  When the
                              machine instruction is invoked
                              in a colon definition, code n
                              will be compiled.
        CREATE                Give the instruction a name.
        ,                     Compile code n in the code field.
        DOES                  Above are compiler action and
                              following are run time function.
        R>                    Get the pointer to the code field.
        77777 AND             Mask off the carry bit.
        @                     Fetch the code n stored in the
                              code field.
        C,                    Now, compile n into dictionary.
                              That is the assembler function.
        ;
```

Most of the NC4000 machine instructions can be defined using uCODE. Here are the words actually defined this way in cmFORTH:

```
100000 uCODE NOP            One cycle Nop.
140000 uCODE TWO            Two cycle Nop.
100020 uCODE SWAP-DROP      Delete N, next item on data stack.
140721 uCODE R>DROP         Delete I, top of return stack.
160000 uCODE @DROP          NOP with a memory cycle.
154600 uCODE 0+c            Adjust for carry.
177300 uCODE N!             Store N to where T points but
                            keep a copy of N on stack.
147303 uCODE -1             Push a true on stack.
104411 uCODE *'             Multiply step.
102411 uCODE *-             Signed multiply step.
100012 uCODE D2*            Left shift the double integer.
100011 uCODE D2/            Right shift the double integer.
102416 uCODE *F             Fractional multiply step.
102414 uCODE /'             Divide step.
102414 uCODE /''            Last divide step.
102412 uCODE *F             Fraction multiply step.
102612 uCODE S'             Square-root step.
147321 uCODE R>             To-R.
157201 uCODE >R             R-from.
147301 uCODE I              Retrieve loop index.
157221 uCODE TIMES          Repeat next instruction.
```

Instructions which use the least significant 5 bits for short literals, internal register numbers, and memory incrementals must compile proper values into this 5 bit field.

```
: -SHORT                    ( -- f )
                            Return a true flag if the current
                            instruction under construction
                            can take a 5 bit short literal or
                            argument.
       ?CODE @ @            Obtain the current instruction
                            whose address is stored in ?CODE.
       177700 AND           Mask off the lower 6 bits.
       157500 XOR           Is it not equal to 157500, which
                            is the code to access internal
                            registers?
       ;
```

Here are some examples that will compile pattern 1575xx in memory to be checked by -SHORT:

```
157504 uCODE MD             Multiplier/divisor register.
157506 uCODE SR             Square-root register.

: FIX                       ( n -- )
```

```
                            Get the 5 bit literal from the
                            instruction pointed to by ?CODE
                            and combine it with n to form
                            a new instruction.  It is then
                            stored back to where ?CODE is
                            pointing to.
        ?CODE @ @           Get the instruction pointed to
                            by ?CODE.
        77 AND              Preserve only the lower 6 bits.
        OR                  OR it into n.
        ?CODE @ !           Store the instruction back to
                            dictionary.
        ;

: SHORT                     ( n -- )
                            Construct an instruction with
                            short literal.  If the instruction
                            cannot accept a short literal,
                            abort with an error message.
        -SHORT              Can the instruction take a short
                            literal?
        IF                  No.
          DROP              Discard n.
          ABORT" n?"        Print error message and quit.
        THEN
        FIX                 Yes.  Include the literal into
                            n and replace the old instruction.
        ;

: @                         ( -- | n -- )
                            A smart @ compiler.  If the address
                            is in the local memory( <32 ),
                            compile a single cycle instruction.
                            Otherwise, compile a regular two
                            cycle memory fetch instruction.
        -SHORT              Is the address in the local
                            memory area?
        IF                  Not in local memory,
          167100 ,C         Compile a two cycle memory fetch.
        ELSE                It is in local memory,
          147100 FIX        Compile a short memory fetch with
                            address as a short literal.
        THEN
        ;                   This is a compiler directive,
                            not a regular Forth @ word.

: !                         ( -- | n -- )
                            A smart ! compiler similar to
                            @?
        -SHORT              Local memory?
        IF                  No.
          177000 ,C         Compile long memory store.
        ELSE                Yes.
          157000 FIX        Compile a short memory store.
```

```
        THEN
        ;


        The NC4000 machine instructions, which must take short literals as
arguments, are compiled directly using SHORT.  Since these instructions are
compiler directives, their arguments or the short literal, must be known at
compile time.  You cannot change the literal or register numbers dynamically at
run time.  In fact, the compiler will abort if you forget to give the proper
argument in the definition.

: I@                        ( n -- )
                            Compile a register fetch instruction
                            to fetch register n at run time.
        147300 SHORT
        ;

: I!                        ( n -- )
                            Compile a register store instruction
                            to store top of stack into register
                            n at run time.
        157200 SHORT
        ;

: @+                        ( n -- )
                            Compile a increment fetch instruction
                            which increments the address by
                            n.
        164700 SHORT
        ;

: !+                        ( n -- )
                            Compile a increment store instruction.
        174700 SHORT
        ;

: !-                        ( n -- )
                            Compile a decrement store instruction.
        172700 SHORT
        ;

: I@!                       ( n -- )
                            Compile a register exchange instruction
                            which swaps contents between T
                            and register n.
        157700 SHORT
        ;
```

7.5. The Compiler Vocabulary

To program in this environment, you will have to be aware of the difference between the compiler directives and regular Forth words, which can be compiled and interpreted. They appear syntactically identical in a colon

word but behave very differently. The compiler directives can only be used in colon definitions and should not be executed outside of a definition. For this reason, all the compiler directives are placed in a special vocabulary named COMPILER and all the regular Forth words are placed in the FORTH vocabulary. In the normal interpretive mode, only the FORTH vocabulary is searched and you cannot access any of the compiler directives. Only when the word : is executed, will the COMPILER vocabulary be made available to the compiler, which will then take advantage of the optimizing compiler and compile efficient machine code whenever possible. At the end of a definition or when an error occurs, the COMPILER vocabulary will be turned off so that you will be protected from the abnormal behavior of these compiler directives.

```
: FORTH                     ( -- )
                            Define the FORTH vocabulary.
       1 CONTEXT !          Deposit hash code 1 in the system
                            variable CONTEXT.  This hash code
                            is used to select the FORTH thread
                            for searching and extension. The
                            thread is stored in the cell immediately
                            above the variable CONTEXT.
       ;

: COMPILER                  ( -- )
                            Define the COMPILER vocabulary.
       2 CONTEXT !          The hash code of COMPILER vocabulary
                            is 2.  It directs the searching and
                            extension to the COMPILER vocabulary.
                            The head of this vocabulary is stored
                            2 cells above the variable CONTEXT.
       ;
```

8. Optimizing Compiler

The compiler in a regular Forth system is very simple. It only has to search the dictionary, find the words and compile their execution addresses. Each word represents one function. The only complication is to build control structures in a definition, which requires additional actions during compilation. The compiler for NC4000 machine is much more complicated due to following reasons:

- The compiler must absorb the functions of an assembler for assembling machine instructions in addition to compiling high level words or subroutine calls.
- More than one function may be performed by a single machine instruction. The compiler must be able to recognize such a sequence of functions and combine them into a single machine instruction.
- There are three memory spaces to be dealt with: the main memory, the local memory, and the registers.
- Deficiency in the prototype chip precluding certain combinations of bit patterns.

In this version of cmFORTH, Chuck Moore chose a very simple and quite effective approach towards optimizing the assembly of machine instructions. He picked three critical points to exercise code optimization: at the end of a definition when ; is executed, whenever a binary ALU code is assembled, and when a shift code is assembled. These three cases cover most situations where optimization is effective. Other situations can be optimized by explicitly hand coding special machine instructions.

An important variable ?CODE is used to control the optimizing process. Whenever a multi-function machine code is compiled, its address is stored in ?CODE so that the smart compiler can work on it. When a high level word (subroutine call), a conditional or unconditional branch, or a loop instruction is compiled, ?CODE is set to zero, in effect turning the smart compiler off for that instruction.

8.1. The Smart ; Compiler

The subroutine call in NC4000 is a one cycle instruction and the subroutine return is an one bit field which can be embedded in many other NC4000 machine instructions. Obviously, if we can recognize the conditions when the return bit can be inserted into the last instruction in a definition, we can always save a machine cycle. Most of the colon definitions can be treated this way by the smart ; compiler.

```
OCTAL                We want to see the bit patterns
                             in machine instructions.  Octal
                             is the most natural representation.

: PACK                       ( a n -- )
                             Pack the return bit into the machine
                             instruction in address a if possible.
                             Otherwise, compile an explicit
                             return instruction.  Terminate
                             the calling word by discarding
                             top of return address.
       160257 AND            These bits are relevant bits which
                                          must be examined.
       140201 XOR            If bits match this pattern, it is
                                       a memory instruction and the return
                                       bit cannot be packed into it.
       IF                    Bit pattern does not match 140201,
         40 SWAP +!          pack the return bit into a.
       ELSE                  Pattern matches with 140201,
         DROP                Discard address a.
         100040 ,            Compile an explicit return instruction.
       THEN
       R>DROP                Work is done.  Exit the EXIT routine
                                     immediately.
       ;

: EXIT                       ( -- )
                             Look through all the possible
                             patterns where the return bit
                             can be packed and pack it.
       ?CODE @ DUP           Last instruction a machine code?
       IF                    Yes.  Go work on it.
         0 ?CODE !           First re-initialize ?CODE.
         DUP @               Fetch the machine code.
         DUP 0<              Is the bit 15 set?
         IF                  Yes.  It looks like a machine
                             code.
           DUP 170000 AND 100000 =
                             Is it an ALU instruction?
```

```
            IF PACK THEN      Yes.  Pack the return bit.
            DUP 170300 AND 140300 =
                              Is it a register fetch instruction?
            IF PACK THEN      Yes.  Pack the return bit.
            DUP 170000 AND 150000 =
                              Is it a short literal store instruction?
            IF                Yes.
              DUP 170600 AND 150000 XOR
                              15x6xx cannot be a valid instruction.
              IF PACK THEN    If not 15x6xx, pack the return
                              bit.
            THEN DROP         End of multi-function code processing.
          ELSE                Last instruction is not a multi-function
                              machine code.  However, if it is a call
                              instruction, it can be substituted by
                              a jump instruction to save an explicit
                              return instruction.
            DUP H @ dA @ - XOR  Compare the address in ?CODE with
                              the current dictionary pointer.
            170000 AND 0=     Are they in the same 4K cell page?
            IF                Yes.
              7777 AND        Isolate the 12 bit address field.
              130000 XOR      Tag the unconditional jump field.
              SWAP !          Store it in the address pointed
                              by ?CODE.
              EXIT            Terminate here immediately.
            THEN
            DROP              Discard contents of ?CODE.
          THEN
        THEN DROP             Discard ?CODE.
        100040 ,              Compile explicit return instruction.
                              Not possible to optimize.
        ;


: ;                           ( -- )
                              The optimizing ; compiler.
        [COMPILE] RECURSIVE   Reset the smudge bit in the name
                              field of the new definition, making
                              it available for searching.
        R>DROP                Exit the compiler loop at the
                              end of this word (;).
        [COMPILE] EXIT        EXIT was made immediate.  Force
                              its compilation.
        ;
```

8.2. The Smart ALU Function Compiler

The ALU instructions are the most complicated type of instructions in the NC4000 chip, because all the fields and bits are functional code. Thus a large variety of instructions can be constructed, doing many things in single machine cycles. A smart compiler would have to be able to recognize all these conditions in order to combine the maximum number of functions into a single machine instruction.

The elementary ALU functions like + , - , SWAP- , AND , OR , and XOR are defined by the smart compiler BINARY. BINARY will examine the instruction previously compiled to see if these ALU functions can be incorporated into that instruction and do so whenever possible.

```
: BINARY                    ( n1 n2 -- )
                            n2 is the code of the ALU instruction.
                            n1 is the pattern which can be XOR'ed
                            into the previous instruction to
                            install the ALU function.
                            Define a smart ALU compiler.
       CREATE               Make a new header.
       , ,                  Compile n2 and n1 into the code
                            field.
       DOES                 Now define what the new compiler
                            directive will do during compilation.
       R> 77777 AND         Pointer to the stored patterns
                            n2 and n1.
       2@                   Retrieve them.
       ?CODE @ DUP          Are we dealing with a machine
                            code?
       IF                   Yes.  Turn on the optimizer.
         @                  The machine instruction.
         DUP 117100 AND 107100 =
                            Is it of the SWAP/OVER type?
         OVER 177700 AND 157500 = OR
                            Or a short literal?
         IF                 Yes.  We can do something with
                            it now.
           DUP 107020 -     Not a DROP?
           IF               Not DROP.
             SWAP-DROP      Discard n2.
             XOR            Force the ALU code into the ALU
                            field of the previous instruction.
             DUP 700 AND 200 =
                            Test if carry must be included.
             IF 500 XOR     If so, restore the Tn bit.
             ELSE
               DUP 70000 AND 0=
                            Make sure we have an ALU instruction
                            at hand, then
               IF 20 XOR THEN
                            flip the Stack Active SA bit.
             THEN
             ?CODE @ ! EXIT The machine code can incorporate
                            ALU code in the ALU field.  Update
                            the machine code.
           THEN
         THEN
       THEN
       DROP                 Drop the ?CODE, which is zero.
       ,C                   Compile n2 as the explicit ALU
                            machine code.
```

```
        DROP                    Discard the compare mask.
        ;
```

```
    Now, all the binary ALU code compiler can be defined
```

by BINARY:

```
6100 101020 BINARY AND
1100 102020 BINARY SWAP-
4100 103020 BINARY OR
3100 104020 BINARY +
2100 105020 BINARY XOR
5100 106020 BINARY -
```

8.3. The Shift Compiler

Shift functions can be appended to all ALU machine code. code. However, restrictions have to be imposed while programming the NC4000 prototype chip so that shifts produce the desired results.

```
: SHIFT                     ( n -- )
                            Define smart shift compilers.
       CREATE               Make new header.
       ,                    Save n, the shift code in the
                            code field.
       DOES                 Actual compilation action.
       R> 77777 AND         Pointer to the stored shift code.
       @                    Get the code.
       ?CODE @ DUP          Is the previous word a machine
                            instruction?
       IF                   Yes.  Do optimization.
         @                  The machine code.
         DUP 171003 AND 100000 =
                            Is it an ALU code without any
                            prior shift operation?
         IF                 Yes.
           XOR              Pack in the shift code.
           ?CODE @ !        Store it back.
           EXIT             Done and out.
         THEN
       THEN
       DROP                 Cannot optimize.  Discard the
                            code address.
       100000 XOR ,C        Compile an explicit shift machine
                            instruction.
       ;
```

The three shift functions which can be safely packed into ALU instructions are:

```
2 SHIFT 2*
1 SHIFT 2/
2 SHIFT 0<
```

One has to be careful about 0< which has to be followed by a NOP before it can be used to do logic branching. As shown in Screen 1, 0< must be redefined for the prototype chip:

```
: 0<                        ( n -- f )
                            Prototype 0<.
       [COMPILE] 0<
       [COMPILE] NOP        A NOP must follow 0< to allow
                            enough time for the bits to
                            propagate.
       ;
```

The double integer shift functions cannot be packed into other ALU code due to the prototype restrictions. They are defined as explicit single cycle instructions:

```
100012 uCODE D2*
100011 uCODE D2/
```

8.4. Merging of DUP

Sometimes a DUP operation can be merged into a machine code, whose stack active bit can be turned on, to accommodate the DUP function. A single cycle DUP instruction must be compiled immediately before the machine instruction under consideration.

```
: DUP?                      ( -- )
                            Pack two previous instructions
                            into one if the first is a single
                            cycle DUP instruction.
       HERE 2 - @           Fetch the instruction just before
                            the one recently compiled.
       100120 =             Is it a single cycle DUP instruction?
       IF                   Yes.  Try to pack.
         HERE 1 - @         Get the most recent instruction.
         7100 XOR           Turn on Tn bit and change data
                            source to T, thus activating DUP.
         -2 ALLOT           Delete the two compiled instructions.
         ,C                 Replace them with a single instruction.
       THEN
       ;
```

Not many instruction pairs can be packed this way. The ones often used in cmFORTH are:

```
: I!                        ( n -- )
                            Compile a register store instruction.
       157200 SHORT         Compile a short literal instruction
                            with n as the register number.
       DUP?                 Often the data stored into a register
                            are needed for other purposes.
                            If a DUP instruction is used this
                            way, it can be packed into the
                            I! instruction.
       ;

: >R                        ( -- )
                            Compile a >R or a DUP >R instruction.
       157201 ,C            Compile the single >R instruction.
       DUP?                 Pack DUP if available.
       ;
```