Footsteps in an Empty Valley

NC4000 Single Chip Forth Engine

Dr. Chen-Hanson Ting

Fourth Edition

Offete Enterprises, Inc.

2017

Preface to the Third Edition

Dr. Glenn Haydon, one of the developers of the WISC (Writable Instruction Set Computer) Forth engine, often made the remark that it was unfortunate that Chuck Moore built the NC4000 chip and cast Forth into silicon. He lost the freedom to change his mind, as he often did. Indeed, it was very difficult to make any change in NC4000. Novix had not been able to fix the bugs in the NC4000 prototype chip, or to bring out the NC6000/5000 with enhanced features, even with the infusion of large amount of venture capital after its incorporation. After two years since NC4000P was released, it is still the only chip available from Novix.

Although Chuck Moore could not do much on NC4000, he is not standing still either. He talked about the 32 bit 'Buffalo Chip' on several occasions, although it is not clear how much has been committed to its design and construction. He also kept on revising the Forth kit and the cmForth system. The latest version was released in December 1987. He sent me a copy of the new cmForth to be published in *More on NC4000* and allowed me to use it in the third edition of this book.

It is a pleasure to go through the new cmForth and compare it line by line with the old version. Chuck made many changes, polishing the code in more than one way. Many definitions are refined. Many names are changed. The style and code layout are also much improved for readability. Shadow screens are included to provide functional description for every word defined. He managed to shave off 15 lines of code and save a whole screen. He also put shades at the right and bottom edges of the text blocks, making them rise above the page. It is rather pleasing to the eyes.

His persistent obsession in pursuing the simplest expression to be shared by NC4000 and its users is always fascinating to me. He must have felt the enormous responsibility of a system programmer and a language designer, in that every instruction he deleted and every cycle he saved will be multiplied by the billions in memory and times saved by the users. The old Chinese master ought to have made this observation when he declared: "For knowledge, add a little everyday. For wisdom (Tao), erase a little everyday". The Tao of Forth is not something vague and ethereal. It is embodied in cmForth for us to see, to feel, to study, and to meditate on, if we cared.

In this edition I tried to include information and developments on NC4000 over the last two years. Many new sections are added based on my papers appeared in *More on NC4000*. However, the major focus of this book is still Chuck Moore's cmForth. Observing Chuck's programming style and how he constructs large structures from the components is the best way to gain maturity in Forth programming. Simplicity manifests itself in correctness, flexibility, capability, and productivity.

In last year (1987), we saw several Forth engines: the 32 bit Forth chip from the Applied Physics Laboratory in John Hopkins University, both a 16 bit and a 32 bit WISC engine from WISC Technology, and the ,16 bit FORCE chip set from Harris Semiconductor. We can expect more entries into this field which will generate more interests and excitement in the coming years. Finally, we start to see blossoms from the seeds Chuck Moore planted two decades ago.

A NC4000 Users Group was formed in the San Francisco Bay Area in 1986. The Silicon Valley

Chapter of the Forth Interest Group hosts the NC4000 Users Group Meeting once every three months, on the forth Saturdays in January, April, July, and October. The discussions in these meetings are always lively and sometimes provocative. For members outside of the Bay Area, a newsletter *More on NC4000* helps the circulation of technical information about NC4000 and related products. Volume 7 of *More on NC4000* was just released. We hope that it will provide timely and useful exchange of ideas and techniques among the users.

Chen-Hanson Ting
San Mateo, California
March 1988

Forward to the First Edition

Footsteps in an empty valley is a Chinese ideographic phrase with very deep poetic connotation. It is used to describe the emotional feeling when one is, about to meet a long missed friend as his footsteps are nearing. The picturesque setting is an empty, desolated valley this person had chosen to retire. He has as his companions the trees and the flowers, a small flowing brook perhaps, and wild lives off the woods--the best mother nature has to offer--except for trusted friends whose friendships he had to sever as the price of his retirement.

One day he was awakened amidst all, the familiar sounds of his environ--whispering of the trees, splashing of water, and soft songs of the birds--by the footsteps of someone he dearly missed all these years. Who would travel this far to this distant valley to visit, but the most intimate and the most trusted of friends?

Living in this Silicon Valley full of people, money, energy, activity, and ideas; we've seen now things invented and new products introduced with rapid pace. Some are successful. Most do not see the light of day. Even the most successful fade in a couple of years. Amongst the high pitched, loud sounding hype, there is always this silent loneliness deep down inside. Which voice shall we heed? What direction are we heading? Where is the best and the truest to be found?

We have witnessed hosts of microprocessors and microcomputers marching from cradle to grave, right before our eyes. Languages and operating systems come and go. Even in Forth, which I use to code for a living and write about to entertain, we've seen good work done and disappear, come and go. Have we the source code. It would be especially helpful to the user when the stack picture got fuzzy and the logic seemed tied in knots. At the very least, the user will have another point of view to explore Chuck's ideas besides reading the source code itself.

The NC4000 is truly a milestone in computer technology-- more so than the much touted RISC computer. The Berkeley RISC machine was essentially a rediscovery of the original Von Neumann's ENIAC design. The only addition was the overlapping registers used to facilitate parameter passing between procedures. NC4000 is much more sophisticated than the RISC machine in its dual stack architecture, single cycle subroutine call and return, and the externally microcoded instruction set. To take advantage of the unique architecture of NC4000 and to make the best use of its powerful instruction set, the user needs a firm grasp on the inner mechanism of this chip. A systematic exposition of this chip is therefore necessary to bring all the information about this chip into sharp focus. Some knowledge about the hardware structure in this chip is mandatory in order to understand the software system embedded in cmForth; although lacking this knowledge would not prevent the user from programming this chip in the normal Forth style. A long section in this book is devoted to the chip itself to provide background information on the chip.

The NC4000 chip does not work by itself. You have to connect it to some RAM and ROM memory to build a computer. Circuit schematics are provided for such a computer so that the user can build it with minimal parts and labor, and be productive in a short time. Many programming tips and tutorial examples are also included. Nevertheless, the richest source of coding examples are in the cmForth source code itself, where the user can find practical solutions to a broad spectrum of problems an

operating system has to solve. There is much we can learn from Chuck Moore both in terms of programming techniques and programming style.

I didn't really have much time to explore all aspects this computer and the NC4000 chip. This manual represents the scope of my understanding at this moment. As time passes, I ¨dill make additions and updates and hope that you will keep me informed of your opinions and suggestions. If the chip is to be successful, it needs the entire Forth community to support it by providing viable applications and services to users not fluent in the language. If it is successful, we will ride on its coattails for a long, long while.

It is superfluous to acknowledge Chuck Moore, because the acknowledgement is implicit every time I utter 'Forth'. However, his personal help in bringing up our first NC4000 system and providing it with cmForth greatly accelerated our pace in making this information available to Forth users. Dr. George A. Nicol and Mr. Scott Reinhart of the Software Composers were very helpful in providing information on their SC1000 computer which also uses NC4000 as its CPU. Mr. John Peters and Dr. and Mrs. Albert Ting read the manuscript and made numerous corrections and suggestions.

My best wishes to you and to your NC4000.

Chen-Hanson Ting
San Mateo, California
March 1986

My Electronic Bookshelf

A couple of years ago, I closed my website www.offete.com and stopped distributing my publications on-line. Nevertheless, these publications still exist on my electronic bookshelf. If you need any of them, please send me a request at chenhting@yahoo.com.tw, I will sent it in a return email, and also bill you by a PayPal invoice. I know, we are in the 21st century now. You cannot do anything without a website. But, at least I got rid of lots of paper, and the snail mail.

Juergen Pintaske twisted my arm to get *Footsteps in an Empty Valley* updated from a printed copy, which was edited on an old word processor TMaker on a CP/M machine and printed with a Diablo daisy wheel printer. Files got lost with the CP/M machine. I had to scan all the pages and used OCR to recover the text. The hardest part was Chuck Moore's source code of cmForth, which he printed on an Epson dot matrix printer with a worn ribbon. Lots of the dots disappeared through copying processes. I tried my best to bring back the code, but couldn't be entirely sure. I hope nobody will use the code for any purpose other than reading.

Well. Let me know if you have any question.

Chen-Hanson Ting,
San Mateo, California
February, 2017

PDF Books

After I learnt a Forth system, I always tried to document it so I could teach other people how to use it. So I wrote about polyForth, figForth, F83, F-PC, and cmForth. When Win32Forth came along, I gave up, because it was too large and too complicated. I then focused on developing eForth for microcontrollers. After retirement, I cleaned out the books off my shelves. People still asked for them, so I converted some to pdf files. Here is the list of available titles:

4001 Footsteps in an Empty Valley, 4th Ed., $15
Description of the first Forth chip NC4000 from Novix, and Chuck Moore's cmForth for it. cmForth was the simplest and most compact specification of a real Forth system for a real Forth computer. It contains a complete Forth system with a target compiler, an optimizing assembler, and a serial disk driver. Required reading for all Forth programmers.

1010 Systems Guide to figForth, 3rd Ed. $15
The most authoritative treatise on how's and why's of the figForth Model developed by Bill Ragsdale. Internal structure of the figForth system. Very detailed discussions on the inner interpreters and the outer (text) interpreter of Forth.

1003 Inside F83, $15
Everything you want to know about the Perry-Laxen F83 system but afraid to ask. 288 packed pages divided into 4 parts: Tutorial on F83 system, Kernel, Utility, and Tools. It is based on 8086

F83 Version 2.1 for the IBM-PC, but useful as a reference manual for all other (8080 and 68000) F83 systems.

1008 F-PC Technical Reference Manual, $15
Narration on all words in the kernel and tools of F-PC, a practically useful Forth system for applications on PC. Functional description of the utilities and applications. Valuable guide to F-PC internals and assembly coding on segmented 80386 architecture.

1013 .eForth and Zen, 3rd Ed. $15
Complete description and exposition of the eForth Model: kernel, high level words, interpreters, compiler and utilities. Comparison of Forth and Zen, their similarities in simplicity and understanding. It is update based on 32-Bit 586 eForth v5.2 for Visual Studio Community 2015. It is in an assembly file as a C++ console project. It uses indirect thread model so that new colon words can be added to the .data segment. It is optimized with 71 code words and 110 colon words.

1015 Firmware Engineering Workshop, $15
A tutorial in 4 parts for building firmware for embedded systems, based on enhanced eForth. Hands-on experiments using CT100 Lab Board with 8051. 8086 eForth 2.02 and 8051 eForth 2.03 are included with the original eForth 1.01 Models for 8086 and 8051.

eForth Implementations

I had always looked for low-cost microcontroller kits to teach people Forth. Over the years, these kits were getting cheaper and more powerful, and I ported eForth to a lots of them. I had lots of fun with them, and I enjoyed seeing others having fun (and making useful products) as well. eForth captures the essence of Forth, as an universal programming language for small, embedded systems. These eForth implementations are distributed with source code and substantial documentation.

2152 ADuC ARM7 eForth, $25
eForth for ADuC7020 MicroConverters from Analog Devices. It is written in ARM7 assembler on a Keil IDE. It uses the ARM7 link register for threading, and is fully optimized to make the best use of ARM7 core and analog peripherals integrated in this true microcontroller.

2153 SAM7 ARM7 eForth, $25
eForth for AT91SAM7X256 microcontroller from Atmel. It is in ARM7 assembler on Keil uVision3 RealView IDE. It uses the DBGU serial port to interact with user. Olimex's SAM7-EX256 Board has a very interest color LCD module. This eForth has graphic primitives to drive the LCD display.

2154 cEF Version 1.0, $25
cEF is a Forth implementation based on eForth Model, and compiled by gcc compiler in Cygwin on a PC. The underlying Virtual Forth Machine has the standard 33 machine instructions defined in the original eForth Model. It is target to microprocessor without floating point coprocessor, and uses only integer arithmetic operations.

2155 cEF Version 2.0, $25
cEF is a Forth implementation based on eForth Model, and compiled by gcc compiler in Cygwin. The Virtual Forth Machine has 64 machine instructions. Multiplication and division are implemented using double arithmetic floating operations. It is highly optimized to take advantages of recent microprocessors with floating point coprocessors.

2157 eForth for STM8S,$25
STM8S is an 8 bit microcontroller from STMicroelectronics. ST is distributing a STM8S-Discovery Board for less than $10. It is an excellent kit to learn microcontroller programming. Now, a good Forth experimental kit is available for high school students.

2159 328eForth for Arduino Uno, $25
This is a very efficient implementation of eForth for ATmega328P microcontroller used on Arduino Uno Kit. It is using Subroutine Thread Model. It uses tools in NRWW memory to compile new words in main RWW flash memory. It allows you to build turnkey systems for commercial applications. It requires a flash programming tool.

2162 ceForth_328 for Arduino Uno, $25
This is an Arduino sketch which can be compiled and uploaded by Arduino IDE. The Forth Virtual Machine is coded in C, and the Forth dictionary is imported as a data array. The Forth dictionary can be extended into the RAM memory, so you can add new commands to this system. The dictionary is produced by a metacompiler running under F#. The source code of the metacompiler is included for you to enhance this system.

2164 430eForth for TI LaunchPad, $25
This is a Forth system for the MSP430G2553 microcontroller used on the LaunchPad from TI. It is a 16-bit Forth implementation to be assembler by the Code Composer Studio 5.2. It makes the best used of the 16 KB of flash memory, leaving about 10 KB for your applications.

2165 STM32eForth720 for STM32 F4 Discovery, $25
This eForth is for STM32F407 chip on STM32 F4 Discovery Kit from STMicroelectronics. This chip has 1 MB flash memory, 192 KB of RAM, and a ton of interesting IO devices. STM32 is no longer an ARM7 chip, but a THUMB2 chip. STMeForth720 is optimized for the new environment.

2166 430eForth v4.3 for TI LaunchPad, $25
This is a Forth system optimized for the MSP430G2553 microcontroller used on the LaunchPad from TI. It is changed from a subroutine threaded model to a direct threaded model, faster and more compact.

2167 8086 eForth Version 2.03, , $25
Enhanced 32-bit eForth for 80586 running under Visual Studio Community 2015. It is assembled by MASM buried under C++ as a console project. Now you can evaluate the eForth model conveniently in latest Windows environment.

2171 32-Bit 586eForth v.5.2 for Visual Studio, $25
It is a assembly file in a C++ console project on Visual Studio Community 2015. It requires library files supplied by Kip Irvine for Windows services. It uses indirect thread model so that new colon words can be added to the data segment. It is optimized with 71 code words and 110 colon words. Now you can test drive eForth on newer Windows PC.

2172 espForth for ESP866 Chip, $25
ESP8266 is a 32-bit microcontroller with integrated WiFi antenna and software drivers. Arduino IDE can compile and upload applications to it. espForth is an Arduino sketch which allows Forth commands to be sent to ESP8266 remotely as UDP packets. IoT for fun!

VHDL Forth Chip Designs

I had used VHDL to design Forth processors and tested them on FPGA's. They included a 16-bit processor eP16 and a 32-bit processor eP32. I ported eForth to these chips for design verification. In 2016, we ran a CPU Design Workshop in Silicon Valley Forth Interest Group, and I used designs of Intel 8080 and DEC PDP1 as exercises. It was interesting that eForth was used here as test benches, which were much more difficult to design than CPU themselves.

2163 eP16 in VHDL for LatticeXP2 Brevia Kit, $25
eP16 is a 16 bit microcontroller. It was implemented on LatticeXP2 Brevia Development Kit with LatticeXP2-5E FPGA. It included a CPU module, a UART module and a GPIO module. An eForth metacompiler producing eForth RAM image is included with all source code.

2158 eP32 in VHDL for LatticeXP2 Brevia Kit, $25
eP32 is a 32 bit microcontroller. It was implemented on LatticeXP2 Brevia Development Kit with LatticeXP2-5E FPGA. It includes a CPU module, a UART module and a GPIO module. An eForth metacompiler producing eForth RAM image. It is the best Forth engine design on the cheapest FPGA kit. All VHDL files and eForth files are included.

2169 80eForth202 for eP8080 Chip, $25
eP8080 was a CPU model used in SVFIG FPGA Design Workshop. It recreated an i8080 chip in FPGA. 80eForth202 was the Forth system embedded in VHDL for design verification and to help debugging the chip. The eForth RAM image was derived from 86eForth v2.2 and Z80eForth by Ken Chen, assembled with MASM.

2170 PDP1eForth for ePDP1 Chip, $25
ePDP1 was another CPU model used in SVFIG FPGA Design Workshop. It recreated a PDP1 chip in FPGA. PDP1eForth was the Forth system embedded in VHDL for design verification and to help debugging the chip. It was derived from eP16, and used a metacompiler in F# to create eForth dictionary to initialize RAM memory.

Contents

Figures

Tables

Chapter1. Introduction

1.1. Historical Background

In the beginning, Chuck Moore invented Forth as a programming language to make himself a more productive programmer. The late 1960's saw Forth evolved into an integrated, unified, and complete software development tool. In 1972, Chuck and a few of his colleagues left National Radio Astronomy Observatory (NARO) where they nurtured Forth into its present form, and formed Forth, Inc. to explore its commercial potential. For a while, he was content to use Forth as a software tool to solve real world problems, leaving hardware engineers to pick up the Forth architecture and implement Forth engines.

Forth didn't become a household name in computer industry until Forth Interest Group was formed in the San Francisco Bay area. figForth source code was distributed by tons at cost beginning in 1978. Although Forth had established a sizable following in the microcomputer user community, the computer industry was very reluctant to accept Forth as an alternative architecture for hardware design and implementation. There were scattered efforts towards 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, founding father of the Forth Interest Group, and John Peers of Logical Machines. Both had intense interests in seeing Forth burnt into silicon. The right milieu was thus gathered for the precise chemistry necessary for brewing a Forth chip.

At the time, Chuck with Glenn Haydon and others were designing a prototype board to execute Forth words as primitive instructions. The spark that initiates the Forth chip development was set by Don Colburn of Creative Solutions, another Forth heavy weight 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 serious support was available.

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 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 Technology Industries, Sysorex International, became interested in the project and formed the Novix partnership in March 1984 to carry forward the hardware implementation of this Forth chip.

Mostek was chosen as the foundry to cast Forth in silicon. The chip was implemented with 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 7 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.

When Mostek was sold to Thomson and Novix went through a process of reorganization and incorporation, efforts in removing the bugs in the prototype mask was suspended. A second run using the same prototype mask was completed, with the pin count on the pin-grid package reduced to 121 pins. It seems that this prototype chip, complete with all its bugs and restrictions, will be the one we have to live with for a while. Even with the bugs and restrictions, NC4000P is more powerful than the best of the 16-bit microprocessors.

In March 1986, Novix was incorporated with Mr. John Peers as its chairman and chief executive officer. It was producing NC4000 chips and selling them. Chuck Moore offered 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 were selling board level products: SC1000C single board computer with NC4000 and 8K of RAM, prototyping PC boards, memory expansion boards, etc. Meanwhile, NC4000 spread to many laboratories, factories, and other countries around the world. It was used in many different areas, including CAD/CAM, data acquisition, fast signal processing, factory automation, artificial intelligence, etc. An NC4000 Users group was also formed to collect and distribute information on NC4000 and its applications.

Novix licensed NC4000 design to Harris Semiconductors to become the 16 bit CPU core in its cell library. Harris called it Forth Optimized RISC Computing Engine or FORCE. Harris' customers can thus use this CPU core together with other supporting macro-cells to design complete single chip microprocessors for dedicated applications. A five chip chip-set was produced by Harris for evaluation purposes. It includes a FORCE CPU, two stack controllers, an interrupt controller, and a multiplier chip. Software Composers, now renamed Silicon Composers, built a coprocessor board with this chip set, to be used inside an IBM-AT microcomputer.

Glenn Haydon and Phil Koopman continued their efforts in building Forth engines using standard TTL parts. In 1986 a 16 bit version was produced, first in kit form and later in printed circuit boards, named CPU/16. This design is especially interesting because it uses writable control store memory to hold the microcode. User can thus design or write his own instructions on this engine. They called it the Writable Instruction Set Computer, or WISC. In 1987, the 32 bit version CPU/32 was released. Both these versions worked as coprocessor board inside IBM-PC or AT computer. Glenn and Phil formed WISC Technologies, and intended to produce microprocessors using these designs.

1.2. The RISC Panacea

The Reduced Instruction Set Computer (RISC) seems to be the fad of computer industry for the 80's. By using a small instruction set, restricting memory access to a few memory fetch and store instructions, and using a large set of register windows, it promises faster execution at lower costs. Will it solve all our computing problems? The RISC architecture was so attractive that many people try to classify NC4000 as a RISC engine in order to give NC4000 an extra polish for selling to the unsuspecting public.

The truth as I see it is that the Forth virtual engine by nature is a Complicated Instruction Set

Computer or CISC structure, because the Forth engine must support an interactive programming environment. A minimum Forth system has at least 250 instructions just to be barely useful. Any real Forth engine is thus bound to be a CISC machine. In fact, the champion CISC machine VAX has an instruction set that matches very well with a Forth virtual engine. The virtue of NC4000 is not that it has a reduced instruction set, but that it can execute its instructions with blazing speed due to simplicity in design and dual stacks supporting very efficient subroutine nesting.

When the reports on RISC machine were published by Dave Paterson in Berkeley, I often compared it to the design of the original von Neumann's Advanced Electronic Machine (AEM) developed around 1946. The similarity is very striking as shown in the two instruction sets in Tables 1.1 and 1.2. AEM had only 21 instructions and RISC had 31. On the other side of the fence, NC4000 has more than 200 valid instructions. Table 1.3 shows only a partial list of NC4000 instructions which can be conveniently named. Comparing these tables, you can see that NC4000 is definitely a CISC machine. Here we shall compare it with the von Neumann AEM and Berkeley RISC to see its true merits.

Von Neumann's AEM was a 40 bit machine designed primarily for numeric computation. It had multiply and divide instructions, but no logic instructions. It addressed only 4096 words of memory. It was a memory oriented machine, in that an internal accumulator provided one operand and was also the destination of arithmetic operations. The other operand was generally fetched from memory. The only other register MQ in the CPU was used together with the accumulator, serving as an extension to the accumulator in the multiplication and division operations. From the contemporary point of view, AEM may seem very primitive, but it was very efficient for scientific and engineering computation.

Let's examine the RISC machine and see how much we have advanced over the last 40 years. The only significant advancements in the RISC machine over AEM are: subroutine call and return instructions, the large number of windowing registers, and instructions operating on registers instead of the accumulator and memory. RISC has a large set of registers because of the VLSI technology, which was not available to von Neumann. Subroutine call and return were invented much later; although von Neumann was keenly aware of the power of subroutines, which was actually coined by him. Conveniently nesting subroutines and returning from them had to wait until the stack was invented in the 50's. Because of the large set of registers RISC has on chip, it is advantageous to use them as much as possible for normal ALU operations, while delegating the memory fetching and storing to special memory instructions.

Besides the efficient subroutine nesting and the use of registers to reduce memory access, RISC is very similar to AEM. It is very interesting to observe that after 40 years of intense research, development, and engineering efforts, we came back to the point where we started. Are we much better than our fathers?

The major advantage of the register windows is that they allow parameters to be passed conveniently between subroutines and their callers. The size of the register window was determined by extensive studies on large compilers and applications. However, no matter how the windows are sized, they tend to be wasteful because most subroutines do not make full use of them, and insufficient on many other occasions. In contrast to register windows, an independent data stack in NC4000 dedicated to parameter passing is the most efficient way to use on-chip memory to support subroutines without limitation on the number of parameters passed into or out of a subroutine.

The major objection to stacks for parameter passing is that stacks traditionally implemented in the main memory cuts into the memory bandwidth. Using data on the stack is thus always slower than using data in the on-chip registers. This objection is no longer valid because large amount of on-chip memory can be dedicated to stacks. It is also possible to build CPU chips which can access external stacks in parallel with the main memory so that stack accessing can be overlapped with memory accessing. NC4000 sports two external stacks in addition to the main memory. One stack is for subroutine nesting, and the other is for parameter passing among subroutines. As the CPU can access the main memory, the data stack and the return stack simultaneously, NC4000 is capable of executing a subroutine call in a single machine cycle and also returning in a single machine cycle.

Table 1.1. Von Neumann Machine Instruction Set

Symbol	Function	Comments
LOAD	S(x)->Ac	Load accumulator
LOADN	S(x)->Ac	Load negative to accumulator
LOADM	S(x)->AcM	Load absolute to accumulator
SUBM	S(x)->Ac-M	Subtract absolute from accumulator
ADD	S(x)->Ah	Add memory to accumulator
SUB	S(x)->Ah	Subtract memory from accumulator
ADDM	S(x)->AhM	Add absolute memory to accumulator
SUBM	S(x)->Ah-M	Subtract absolute memory
LOADR	S(x)->R	Copy memory to register
MOVR	R->A	Copy register to accumulator
MUL	S(x)*R->A	Multiply memory with register product in accumulator-
DIV	A/S(x)->R	Divide accumulator by memory
JMPL	Jump S(x) Left	Jump to the left address at x
JMPR	Jump S(x) Right	Jump to the right address at x
BRAL	Branch S(x) Left	Jump to left address at x if A>=0
BRAR	Branch S(x) Right	Jump to right address at x if A>=0
STR	A->S(x)	Store accumulator to memory
STRL	A->S(x) Left	Store left half of accumulator
STRR	A->S(x) Right	Store right half of accumulator
SHR	R	Arithmetic right shift in accumulator
SHL	L	Double arithmetic shift of the accumulator-register pair

Table 1.2. RISC Instruction Set

Symbol	Function	Comments
ADD	Rd<-Rs+S2	Integer add
ADDC	Rd<-Rs+S2+carry	Add with carry
SUB	Rd<-Rs-S2	Integer subtract
SUBC	Rd<-Rs-S2-carry	Subtract with carry
SUM	Rd<-S2-Rs	Reverse subtract
SUBCR	Rd<-S2-Rs-carry	Reverse subtract with carry
AND	Rd<-Rs&S2	Logical AND
OR	Rd<-Rs;S2	Logical OR
XOR	Rd<-Rs xor S2	Logical exclusive OR
SLL	Rd<-Rs shifted by S2	Logical shift left
SRL	Rd<-Rs shifted by S2	Logical shift right
SRA	Rd<-Rs shifted by S2	Arithmetic shift right
LDL	Rd<-[Rx+S2]	Load long
LDSU	Rd<-[Rx+S2]	Load short unsigned
LDSS	Rd<-[Rx+S2]	Load short signed
LDBU	Rd<-[Rx+S2]	Load byte unsigned
LDBS	Rd<-[Rx+S2]	Load byte signed
STL	[Rx+S2]<-Rm	Store long
STS	[Rx+S2]<-Rm	Store short
STB	[Rx+S2]<-Rm	Store byte
JMP	pc<-Rx+S2	Conditional jump
JMPR	pc<-pc+Y	Conditional relative jump
CALL	Rd<-pc, next,pc<-Rx+S2, CWP<-CWP-	Call and change window
CALLR	Rd<-pc, next, pc<-Rx+Y, CWP<-CWP-	Call relative and change window
RET	pc<-Rm+S2, CWP<-CWP+1	Return and change window
CALLINT	Rd<-last pc, next, CWP<-CWP+1	Call and disable interrupts
RETINT	pc<-Rm+S2, CWP<-CWP+1	Return and enable interrupts
LDHI	Rd<31:13><-Y,Rd, Rd<12:0><-0	Load immediate high
GTLPC	Rd<-last pc	Restart delayed jump
GETPSW	Rd<-PSW	Load status word
PUTPSW	PSW<-Rm	Set status word

This capability of single cycle subroutine call/return is very significant, in the light of the studies the Berkeley RISC group made to justify the RISC architecture--that subroutine calls and returns often consume 40% of the CPU time in high level languages and compiler implementations. Minimizing subroutine call and return will thus have a very significant impact on the efficiency of large applications programmed using high level languages.

Another interesting feature of NC4000 is that the address generation was given the highest priority in the CPU design. Consequently the addresses of the next memory locations, be them in main memory or in the external stacks, are always made available midway through a machine cycle. The program can thus branch forward or backward, conditionally or unconditionally, in a single cycle. This design solves the problems generally associated with RISC architectures which have to rely on an instruction

pipeline to achieve the goal of one instruction per cycle.

NC4000 has the following program flow control instructions at the machine level:

Subroutine Call
Subroutine Return Begin ... Until
Begin ... While ... Repeat
Do ... Loop
If ... Then ... Else
Jump
Conditional Branch

They can be used to support all high level languages which require these control structures.

NC4000 with its dual stack architecture to support fast subroutine calling and returning, single cycle execution of instructions without pipelining, and the support of structured programming languages at the machine code level is definitely a superior design than the Berkeley RISC machine. The RISC panacea is not in the reduction of number of instructions, but in the reduction of the complexity of the CPU and the logic structure inside the CPU.

Table 1.3. NC4000 Instruction set

Stack Instructions	`DUP DROP OVER SWAP NIP NIP-DUP DROP-DUP OVER-SWAP >R R> R@`
ALU Instructions	`+ - +c -c AND OR XOR SWAP`
Compound Instructions	`OVER-aluop OVER-SWAP-aluop SWAP-OVER-aluop 2DUP-aluop`
Shift Instructions	`2/ 2* D2/ D2*`
Special Arithmetic Instructions	`0< *' *- *F /' /" S'`
Memory Instructions	`@ ! @+ !+ @- ! - I@ I! I@! LITERAL SHORT-LITERAL Local-memory-fetch Local-memory-store`
Control Structure Instructions	`CALL RETURN IF ELSE #LOOP TIMES`

Chapter 2. The NC4000 Chip

2.1. 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. The dual stack architecture greatly reduces the overhead of subroutine implementation and makes NC4000 especially suited to support high level languages other than Forth. A number of distinguishing features of this Forth engine on silicon can be 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 permitting 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 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.

2.2. External Data Paths

NC4000 chip is housed in a 121-pin, pin grid array package. The pin layout is shown in Figure 2.1. The names and function of the pin groups are shown in Table 2.1.

The external data paths spawn by the large number of pins can be shown schematically in Figure 2.2. 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. The detailed properties of these pin groups are discussed in the following subsections.

(View from top of the pin grid array)

	13	12	11	10	9	8	7	6	5	4	3	2	1
N	B00	A00	A01	A04	A06	X01	X02	X03	J01	J03	J06	S00	S01
M	B01	R01	R00	A02	A05	X00	VDD	X04	J02	J05	J07	D00	D02
L	R03	B02	VSS	WEB	A03	A07	VSS	J00	J04	WER	VSS	D01	S03
K	B04	B03	R02								S02	D03	D04
J	B05	R05	R04								S04	S05	D05
H	R07	B06	R06								S06	D06	S07
G	B07	VDD	VSS							°	VSS	VDD	D07
F	R08	B08	R09								S09	D08	S08
E	B09	R10	R11								S11	S10	D09
D	B10	B11	R13							INDEX	S13	D11	D10
C	R12	B13	VSS	B15	A10	A14	VSS	K01	K05	D15	VSS	D12	S12
B	B12	B14	WED	A09	A12	A15	VDD	K00	K03	K06	S15	S14	D13
A	R14	R15	AOS	All	A13	RST	INT	CLK	K02	K04	K07	WES	D1d

Figure 2.1. NC4000 Pin Layout.

Table 2.1. NC4000 Pin Names and Functions.

Pins	Function
AO-A15	Main Memory Address Bus
BO-B15	I/O Port Bus
CLK	Processor Clock Input
INT	External Interrupt
JO-J7	Return Stack Address Bus
KO-K7	Data Stack Address Bus
DO-D15	Main Memory Data Bus
RO-R15	Return Stack Data Bus
RST	Processor Reset
SO-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

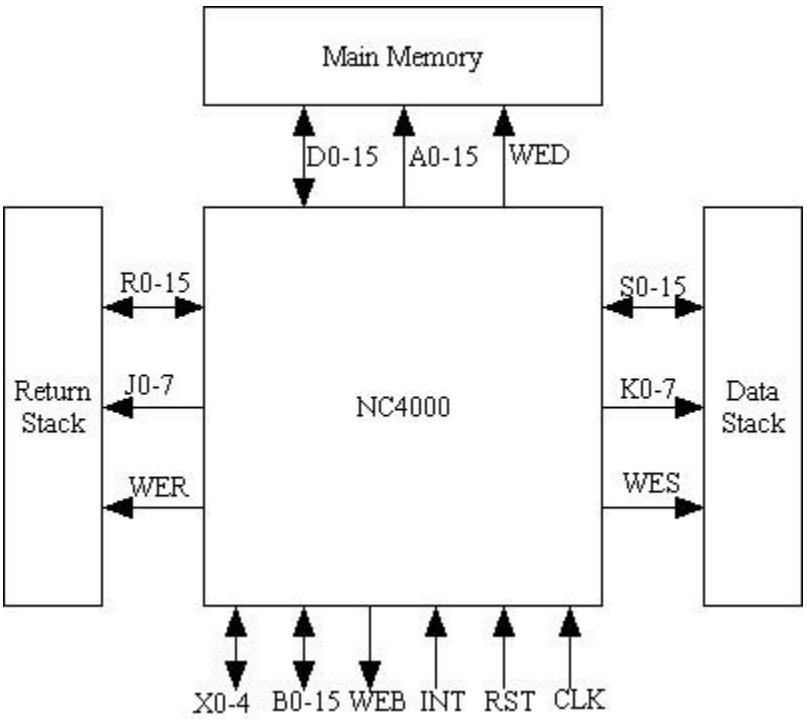


Figure 2.2. External Data Paths of NC4000

2.2.1. Main Memory

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 CLK. 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 stabilized 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 limited by the time NC4000 requires to calculate the next address during the high period of the clock, and the time required by main memory to put valid data on the data lines during 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 NC4000 can address 64K words of memory, only the lower 32K can be used as program memory because the MSB bit of an instruction is a flag 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 NC4000 with single word instructions, while other memory locations must be accessed by explicit memory instructions. Hence the memory starting at 0 is preferably RAM memory if the software is to take advantage of this hardware function. The memory map of NC4000 is shown in Figure 2.3.

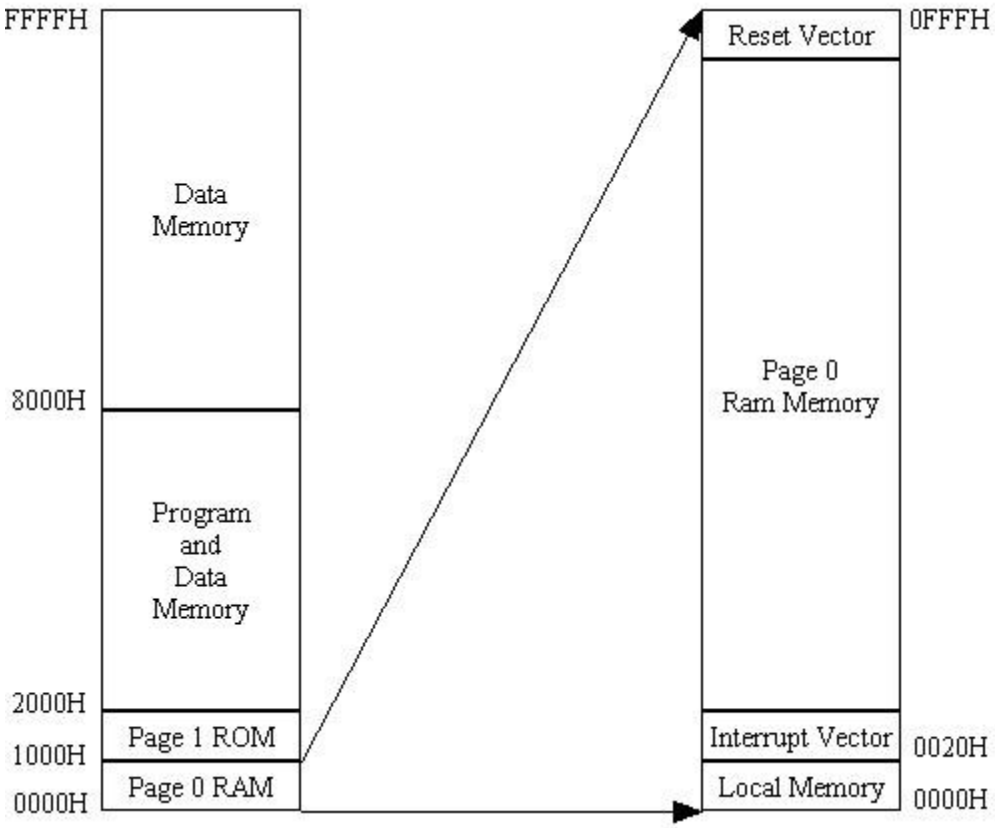


Figure 2.3. NC4000 Memory Map

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.

2.2.2. Data Stack and Return Stack.

A Forth engine 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 3 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 current word would be overwritten.

The depth of the stack is generally much more than sufficient because most programs use a depth of only 12 words each on the return and data stacks. The depth of the stacks will be a serious concern only when a recursive procedure is used. Care must be taken to avoid exceeding the depth of the stacks.

The timing requirements for stacks are almost identical to those of 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 stabilized when the clock goes from high to low. The stack memory must put the data requested on the stack data lines before the clock returns from low to high. Thus the same type of memory used in main memory can be used for stacks. There is little advantage to use higher speed memory for 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 capacities greater than 256 bytes, it seems rather wasteful to use them for stacks in NC4000. One way to take advantage of the extra stack space is to use the lines in the X-port for bank switching of the stacks. This is very useful in supporting a multi-tasking 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.

2.2.3. B-Port and X-Port

Two input/output ports are supported by 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 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 stabilized. 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, you can invert the input data on the fly as they are read through the data register. This extra XOR logic operation is programmable for each individual I/O pin.

Each I/O pin is capable of sourcing or sinking 60 mA, so that they can be used to drive a large number of logic gates without additional buffering chips.

2.2.4. System Timing and Control

NC4000 is an asynchronous CPU chip which requires a single phase master clock. All internal registers are static and the information held in them remains indefinitely while the clock is held low. The upper 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 data 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 timing diagrams of NC4000 are shown in Figure 2.4. Most NC4000 instructions execute in one machine cycle. When the clock line makes a low to high transition, the current machine instruction as well as all input data are latched. The instruction is decoded and executed. When the clock line makes a high to low transition, address of the next instruction is available on the main memory bus and stack addresses are also stabilized on their respective buses. The external memory and stack must put their data on the data lines so that when the clock line is again raised to high, the next instruction and other data will be ready for NC4000 to use.

Memory accessing instructions need two machine cycles to complete. At the end of the first cycle, the address of the memory to be read or written is put on the main memory address bus. The first half of the second cycle is used to read or write the data to or from the main memory. The address of the next instruction will be ready when the clock is lowered. Main memory must then supply the next instruction before the clock is raised again.

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. The external reset circuitry must be capable of sinking 60 mA to bring down the RST line for NC4000 to reset.

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 severely restricted, because the interrupt can be serviced only when a single cycle, non-jump 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.

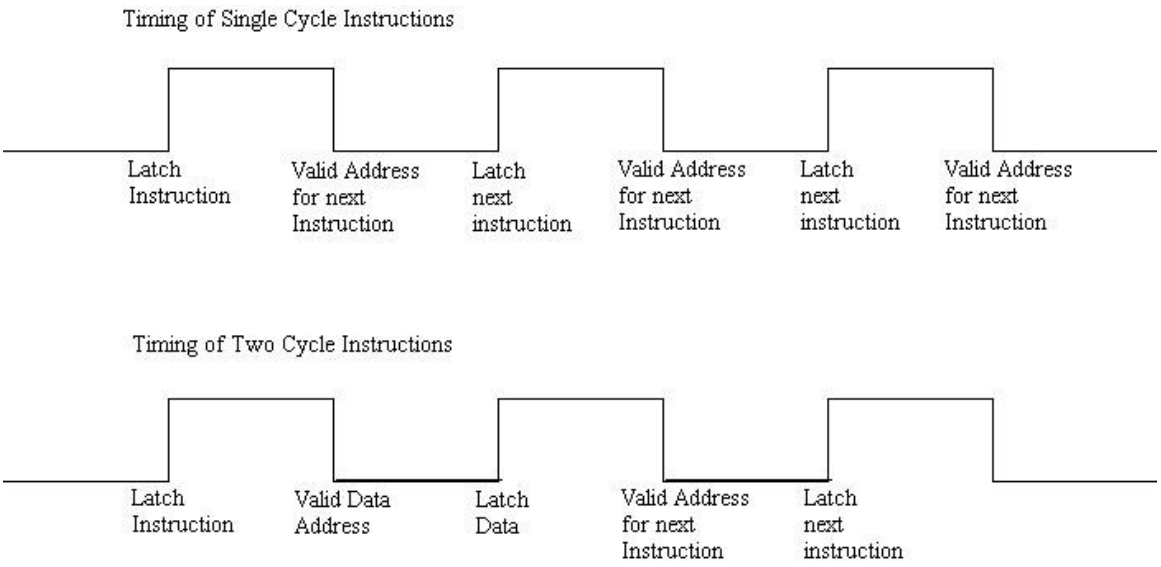


Figure 2.4. Timing Diagrams of NC4000

2.3. NC4000 Architecture

There are two unique features in NC4000 chip which differ from conventional microprocessor design. The chip can simultaneously address four memory spaces--main memory, data stack, return stack, and internal registers including I/O ports. It can directly decode the bit patterns in instructions and execute them in single cycles. Due to the use of two stacks, which greatly simplified the language and the architecture of the chip, a high level language was cast in silicon using only 4000 gates while achieving a speed far in excess of those microprocessors with much more complicated structures.

2.3.1. Internal Registers

The architecture of NC4000 is shown schematically in Figure 2.5. It can be divided into five functional groups internally. On top are the main memory interface and the program control section that fetch instructions and data from the main memory, decode the instructions and execute them. Address Multiplexer A outputs addresses to the main memory, and the data to/from the data memory is latched into Main Memory Port M. Instructions are then copied into the Instruction Latch L for decoding and execution. Program Counter P keeps the address of the next instruction and feeds it to the Address Multiplexer.

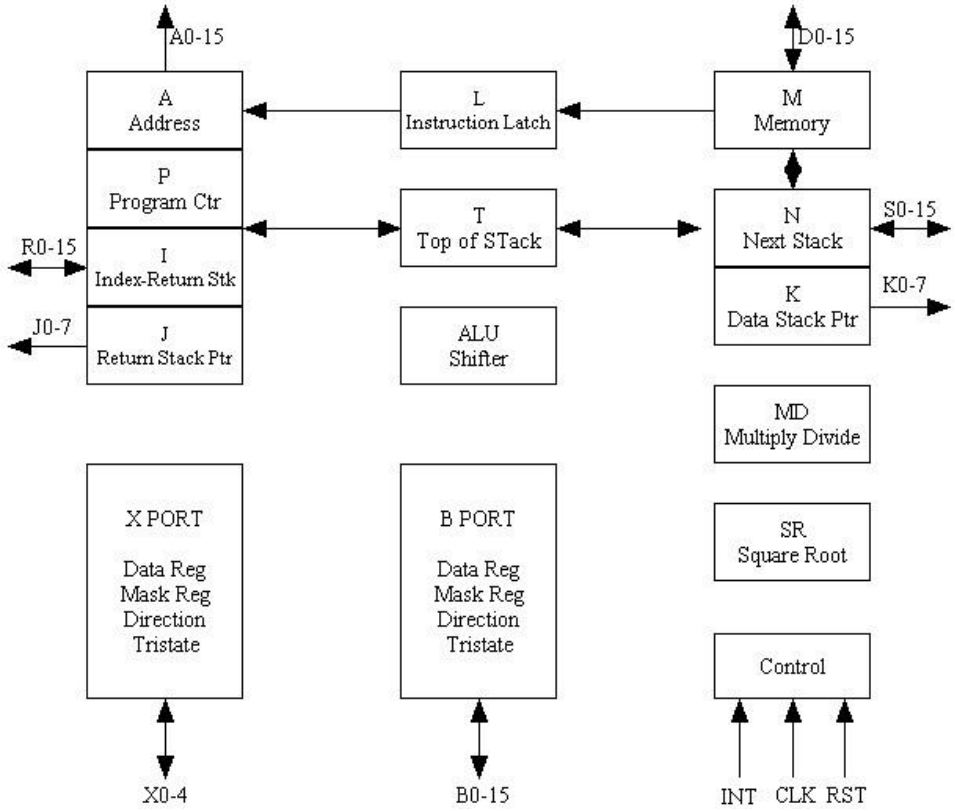


Figure 2.5. Architecture of NC4000

In the middle is the stack interface and control, which manages the data stack and the return stack. The bulk of the two stacks are implemented using off-chip RAM memory. Only the top two elements of the data stack and the top element of the return stack are kept on chip. The Top Register T of the data stack is the focus of the entire chip because it can communicate with almost all the on-chip registers. The Next Data Register N communicates with the external data stack and also receives data from the Main Memory Port. The Return Index Register I is the top of the return stack, doubling as the loop index register. The Stack Pointer Register J/K outputs stack addresses to both stacks. Its upper byte contains the data stack address and its lower byte contains the return stack address.

At the lower center is the ALU section, which performs all the arithmetic and logic operations required by the language. The sources of the ALU are the T and N registers on the top of the data stack. A shifter below the ALU can shift the ALU output before storing it back to T register. The ALU can alternately use the MD or SR registers as input instead of the N register to do multiply/divide or square-root operations. When MD and SR registers are not used for these operations, they are free to be used as scratch pad registers to store temporary data.

Table 2.2. Internal Registers in NC4000

Symbol	Function
A	Address Multiplexer for the Main Memory. It selects data from P, L, I, or T.
I	Top of the return stack. It also serves as a decrementing counter for NEXT(LOOP) and TIMES instructions.
J	Return Stack Pointer Register.
K	Data Stack Pointer Register.
L	Instruction Decode Register. It generates appropriate control signals to execute an instruction.
M	Main Memory Data Bus Port providing data path to main memory.
N	Next Register on the data stack. It can be shifted with T register under instruction control.
P	Program Counter pointing to the next instruction in the main memory.
R	Return Stack Memory Bus.
S	Data Stack Memory Bus.
T	Top Register on the data stack. It performs all ALU functions and communicates with all other registers.

On the lower left are the I/O ports and control registers. There are two fully programmable I/O ports--a 16 bit B-Port and a 5 bit X-Port. Each port is controlled by 4 registers: a Data Register to hold input or output data, a Direction Register to specify individual pins as input or output, a Mask Register to deactivate output pins, and a Tristate Register to tristate output pins outside the output write cycles. These registers give you total control over the 21 I/O pins.

Finally, a block of logic handles the external clock input, the reset RST signal, and the interrupt INT signal. There is a flip-flop in the interrupt circuitry. A high-to-low edge on the INT pin sets the flip-flop and an interrupt return resets it. The flip-flop is set even when the interrupt is disabled. Interrupt, when enabled, generates a subroutine call to memory location 20H. RST causes NC4000 to execute the instructions starting at 1000H.

2.3.2. Program Sequencer

Programs are sequences of instructions stored in the main memory. Normal program execution sequence for NC4000 is to fetch an instruction from the main memory by placing the address of the instruction in the Address Multiplexer or L register before the system clock's trailing edge. During the low period of the clock cycle, the main memory puts the next instruction on the data bus and it is latched into the Main Memory Port or the M register at the rising edge of the clock. The instruction is sent into the Instruction Latch L to be decoded and executed. The first step in the decoding process is to find the address of the next instruction. Normally the next instruction is in the next memory location pointed to by the Program Counter P. In this case, the Program Counter will send its content to the Address Multiplexer and the address of the next instruction will appear on the address bus when the clock level makes a high to low transition. The next instruction will then be fetched and so on.

When a branch or a loop instruction is encountered, the address of the next instruction can be constructed very easily by attaching the upper four bits in the Program Counter P to the lower 12 bits

in the branch or loop instruction. The resulting address is then sent to the Address Multiplexer to alter the program sequence. This is the reason why NC4000 can branch or loop only within the current 4K word page. No computation is necessary to generate the target address. It assures that the next address is stabilized before the clock falls again.

When a subroutine call instruction, characterized by a zero in the most significant bit 15, is executed, the address of the callee appears as the lower 15 bits of the call instruction. This whole instruction is then copied into the Address Multiplexer to fetch the next instruction. In the mean time, the address of the next instruction in P is pushed onto the return stack and copied into the Return Index Register I. When the subroutine is completed, a return instruction will be executed. The return instruction causes the return stack to pop its top item, which is in the I register, back into the Program Counter P and the Address Multiplexer A. Consequently, the caller routine will continue on from the point interrupted by the subroutine call.

When the program counter is push on to the return stack, the most significant bit of the address is of no practical use. Chuck Moore chose to save the carry bit at this position with the 15 bit return address. It is important to mask this carry bit when the return address is retrieved.

When a two cycle memory reference instruction is executed, the memory address, which is usually in register T, is selected by the Address Multiplexer and put on the address bus. The next rising edge of the clock will latch the data from the specified memory address into the N register through the Main Memory Port. Meanwhile, the address of the next instruction is placed on the address bus through the Address Multiplexer and the next instruction will be available at the next rising edge of the system clock.

A unique feature of the Instruction Latch L register is that the instruction latched into this register can be repeatedly executed as many as 65535 times by preceding the instruction with a TIMES instruction. TIMES places a count in the I register which causes the next instruction latched in the L register to be repeated that many times. The most obvious use of this feature is in the construction of multiply, divide, and square-root functions by simply repeating the multiply, divide and square-root step instructions. It is also useful in moving blocks of data. Since a single NC4000 instruction can perform many Forth functions, this repeating capability can be very powerful in various situations.

2.3.3. Data Stack and Return Stack

The top of data stack is the heart of a Forth engine because all data manipulations occur in these few locations. In NC4000, the top two elements of the data stack are cached on chip in the form of the Top Register T and the Next Data Register N. These two registers usually supply two arguments to the ALU unit. T register can communicate with almost all internal registers through internal register fetch and store instructions. The N register also serves as the interface to the external data stack and to the main memory port M, in addition to supporting the ALU unit.

Only the top element of the return stack is cached on chip as the I register. The main purpose of the I register is to interface with the program counter P and the address latch A during subroutine calls and returns. For this purpose, a single cached element is quite sufficient. In addition to this role, the I register has two explicit functions--to hold the loop index in the do-loop structure, and to hold the

count for TIMES instruction. In these cases, the content of the I register is automatically decremented when the NEXT (or LOOP) instruction or the instruction latched in L register is executed. The do-loop or the latched instruction will be repeated until the count in the I register is decremented to zero. At this point, the return stack is popped back to I, and the next instruction in the normal sequence is fetched and executed.

The loop structure thus implemented is quite different from the standard DO-LOOP structure, which requires both the loop index and the index limit to be pushed on the return stack. However, the single decrementing index is sufficient for all looping structures. Chuck Moore stated: "I apologize for having mislead you for so long with DO-LOOP, but the decrementing FOR-NEXT loop is the right way to do it after all". Many Forth programmers have already adopted this simpler loop structure in their systems based on other microprocessors.

When a subroutine call instruction is executed, the address of the subroutine is extracted from the instruction latch L and passed to the Address Multiplexer A, in preparation for jumping to the subroutine. Meanwhile the content of the Program Counter P, pointing to the address of the instruction right after the subroutine call instruction, is pushed on to the return stack. This pushing action involves moving data from P register to I register, and pushing the I register onto the off-chip return stack. When a subroutine return instruction is executed, the content of I register is moved into the A and P registers, and the top of the external return stack is popped into the I register.

Two stack pointer registers J and K are on-chip to control access to the external data stack and the external return stack. As both stack pointers are only 8 bit wide, they are combined into one register J/K when accessed as an internal register. J, the return stack pointer, appears as the lower byte in the J/K register. K, the data stack pointer, appears as the high byte. During a push operation, the stack pointer is pre-decremented, and during a pop operation it is post-incremented.

The data stack, the return stack, and the three stack registers in NC4000 chip can be viewed as an array of 515 words or a 515 element shift register. The entire array can slide right or left with a three word window exposed to the ALU. Elements can be modified, added, or deleted only within this three element window -- most frequently through the Top Register T.

The dual stack architecture contributes greatly to the simplicity of the Forth language and also of NC4000 chip, because it allows the return addresses to be stored independently from the data to be passes between caller routines and the callee subroutines. Thus a subroutine call instruction only has to manage the address on the return stack and leaves the caller and callee routines to worry about the data passing through the data stack. In NC4000, the overhead in supporting the subroutine call and return is reduced to a minimum of one machine cycle, a truly monumental breakthrough in computer design.

2.3.4. Arithmetic Logic Unit (ALU)

The ALU in NC4000 is a 16 bit, dual input Arithmetic/Logic Unit. It is shown in Figure 2.6 with the registers directly connected to it. The Top Register T is always an input to the ALU. It also receives output from the ALU. The other input Y can be taken from the N register, the Multiplier/Divisor register MD, or the Square-Root register SR. The carry bit can be taken as input optionally with the

N register. There is a multiplexer used to supply data from the selected register to the Y input. The multiplexer is controlled by a two bit Y field in the ALU instruction.

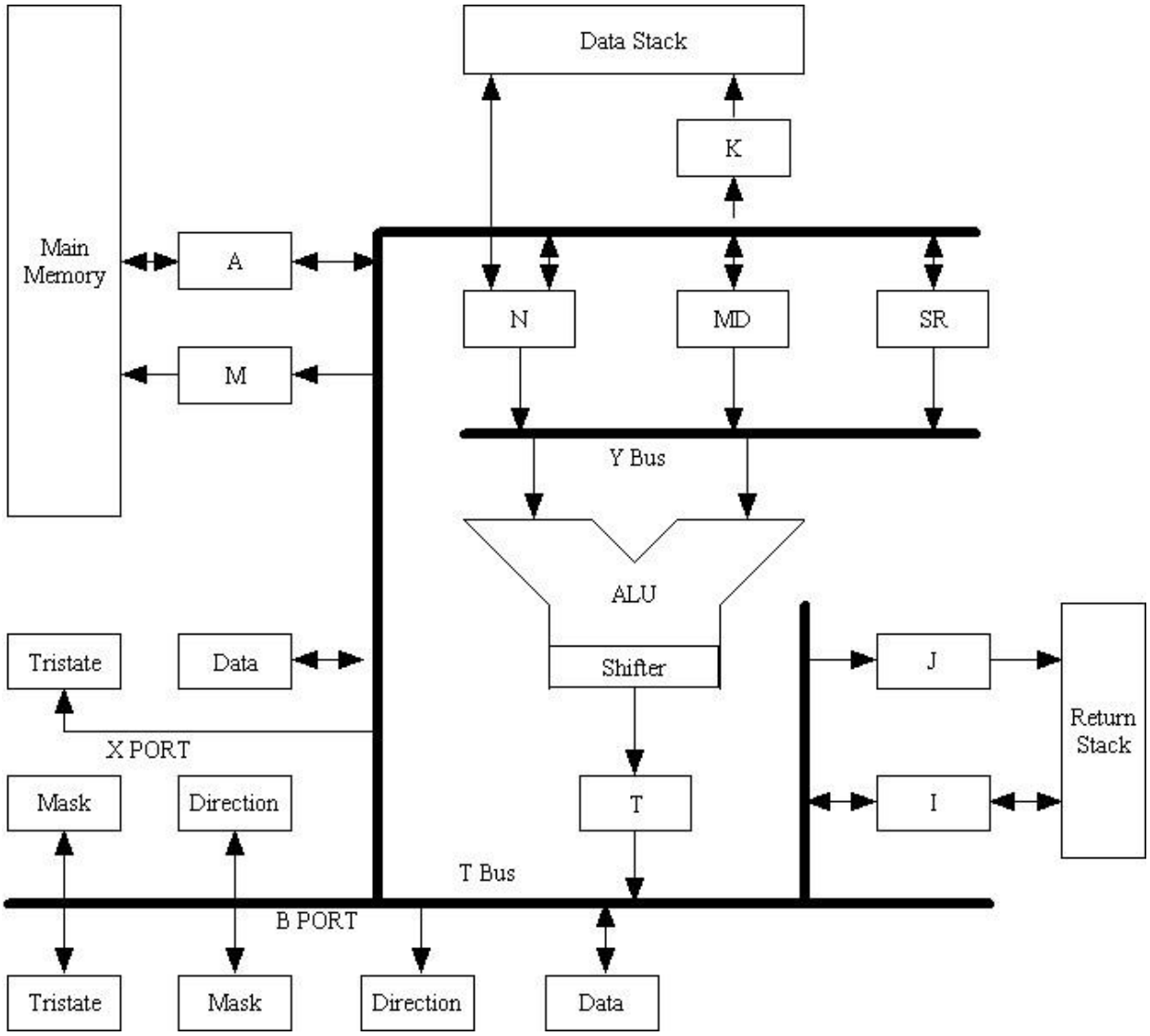


Figure 2.6. Arithmetic Logic Unit

The output of ALU passes through a shifter before falling into the T register. This shifter can shift the 16 bit result from the ALU either left or right by one bit. It can also extend the sign bit of the result to the lower 15 bits. Alternately, it can also shift the 32 bit combination of T and N register pair either right or left by one bit. The shifting function is controlled by the last three bits in the ALU instruction: D?, SL, and SR.

Output from the ALU always goes into the T register. Original content of T register may be copied into the N register. The T Lo N data path is opened or closed according to the bit TN in the ALU instruction. While all these activities are going on in the ALU section, the data stack can also participate in some of these activities. Another bit SA in the ALU instruction allows you to specify

whether the data stack should be pushed or popped, thus passing data between the N register and the external data stack. Permutation of these bits in the ALU instruction produces a rich set of primitives executing multiple Forth instructions in a single machine cycle.

The detailed operation of the ALU, the interaction of the ALU operations, and data path selections will be elaborated in the next chapter dealing with NC4000 instruction set.

2.3.5. The I/O Ports

There are two I/O ports on NC4000 chip: a 16 bit B-Port and a 5 bit X-Port. All 21 bits are connected to pins on the pin-grid package. All the pins are fully programmable to be input, output, or tristate, thus allowing NC4000 to communicate with a host of standard or custom peripheral devices.

The data paths in the I/O section are rather straightforward. There are four port registers for each port; a Data Register which connects to the pins to send or receive data to the outside world, a Direction Register to specify whether pins are input or output, a Mask Register to indicate which pins can be written to, and a Tristate Register to indicate whether the data on output pins are latched or tristated after being written to. All these registers are connected to the T register, which downloads data to any register by register write instructions and reads data from any register by register read instructions. All 16 bits in the B-port registers are used in controlling the 16 B-port I/O pins. Only the lower 5 bits in the X-port registers are used to control the 5 X-port pins.

In the Direction Registers, a bit set indicates the corresponding pin is an output pin, and a bit reset indicates an input pin. In the Mask Registers, a bit set indicates that writing to the corresponding pin is prohibited. In the Tristate Registers, a bit set indicates that the corresponding output pin will go tristate after the write cycle. The Mask and Tristate register settings do not have any effect on input pins.

The Data Registers are used to output data patterns to the output pins and to read input data from the input pins. When a pin is assigned as input, the corresponding bit in the Data Register can still be written to as a comparison latch. Actual bit read from the Data Register is the XOR'ed result of the data at the input pin and the bit written to the comparison latch. If a 1 is written to an input bit in the Data Register (setting its comparison latch) and a 1 appears on the input pin, the result read from the Data Register will be 0 instead of 1. Thus a free XOR logic operation can be performed on the input data on the fly at no cost to the user.

As each input and output pin can typically sink or source 60 mA of current at about 20 pf capacitance, NC4000 has more driving power than most of the available CMOS interface driver chips. It can comfortably drive any CMOS or TTL peripheral chip with plenty of margin. Actually, the B-Port is designed to drive a heavily loaded bus structure, as B stands for Bus.

The designed purpose of the X-port is to support extended memory space addressing. Two special instructions X@ and X! put a 5 bit literal out on the X-port when the master clock goes low. Thus this 5 bit pattern can be used by the main memory as a page select to address up to 32 pages of 64K word memory. Alternately, this 5 bit pattern can be used to select a page in the data stack/return stack space, thus allowing the stacks to be switched between different tasks to support multiuser and

multitasking operations. In the current NC4000P prototype chip, X@ and X! instructions do not work as designed. Thus the X-port pin can only be used to select memory banks statically, not dynamically in single machine cycles.

Chapter 3. Instruction Set of NC4000

Let's consider how a user interacts with a conventional computer--the different layers he has to go through between issuing a command and actually performing the demanded function. The command is first issued to the operating system, which calls a compiled set of instructions into memory for execution. The program usually consists of a set of statements written in either a high level language or in assembly language. This program is compiled or assembled into a set of machine instructions. When the machine instructions are executed, microcode inside the CPU are invoked to do the dirty work of operating the gates and shuffling the data bits. Thus there are at least 5 levels of interpretation between you and the real action. It is a miracle that the computer works at all through this convoluted process. NC4000 architecture reduces the layers to only two levels--user commands and machine instructions. This greatly reduced complexity is the most important reason for the speed and the versatility of NC4000.

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 NC4000 would take individual bits in the instruction and perform individually assigned functions in parallel. It does not need another layer of microcode to perform functions of an instruction. The side benefit is that many Forth instructions can be encoded in one instruction and executed in a single machine cycle.

There had been many projects implementing Forth engines in hardware. All these designs had 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 in NC4000. He attempted was to find the simplest way to control stacks and perform operations while using the minimum number of gates. He discovered that many of the computing and controlling 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 computer. They do not have to be performed serially. These distinct, almost independent domains can be controlled by the limited number of bits in a 16 bit instruction just as well as multiple bits in a much wider microcode. Thus it is possible to perform several functions in a single machine cycle rather than using multiple machine cycles to perform a single function, as implemented in all conventional microcode based computers.

Under the cmForth operating system or other Forth for NC4000, NC4000 engine directly executes normal Forth words or programs just as any other computer operating under a Forth operating system. A user is not really required to know the detailed structure and the machine instruction set of NC4000 in order to use it. However, speed and efficiency can be maximized if the programmer is aware of the special properties of NC4000, its instruction set, and the best way to program it, especially in time sensitive applications. In this chapter, we shall discuss this instruction set in details. Learning the instruction set is the best way to appreciate the power and versatility of this processor. It is also important to understand NC4000 instruction set in order to study the code in cmForth, a piece of art in software by the master himself.

3.1. Classification 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 an instruction is encoded in the most significant four bits of the machine instruction, as shown in Figure 3.1.

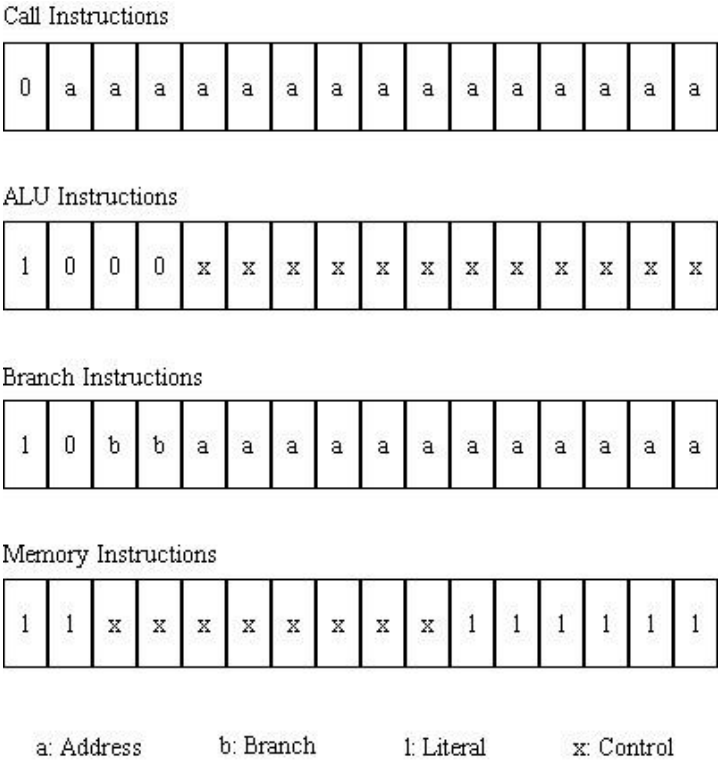


Figure 3.1. Encoding of NC4000 Instructions

Bit 15 is truly the most significant bit in 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 in the main memory. The upper 32K word addressable memory cannot be used to store executable programs. This was a very serious trade-off in the design of NC4000. The most important argument to justify this trade-off is that Forth programs written for NC4000 can be extremely compact due to the single cycle subroutine calls and the condensation of many Forth words into a single instruction. Many large programs are needed to fill up the 32K word program space. By the time memory requirements exceed 32K words, you will probably have a 32 bit Novix chip to accommodate these stupendous programs resulting from the lazy-minded 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.

Another way of classifying NC4000 instruction set is shown in Figure 3.2. 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.

In all I/O, memory, and ALU instructions, types 0 and type 4-7, bit 5 is called the return bit, or ';'. 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; it gets a free ride when the last instruction in a subroutine is an IO/Memory or ALU instruction. It would be difficult to optimize a subroutine return instruction much further.

In the prototype NC4000P chip, however, this return bit should not be tagged to a two cycle memory instruction, types 4-7, 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 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 determine the function of the ALU section on the chip. Thus a free ALU operation can be tagged on to an IO/Memory operation.

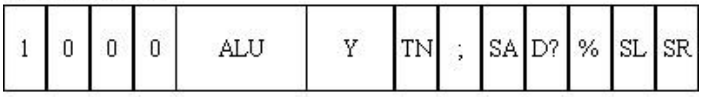
An ALU operation requires two operands. One of the operands is always taken from the Top register T and the other operand is usually selected among three internal registers, specified by a two bit field Y, bits 7 and 8. In most instances, the Y field is zero, which selects the Next register N as its 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. If TN bit is set, the content of T is copied 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 content 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 NC4000 to perform most Forth ALU operations and stack operations, as well as their combinations.

In a memory instruction, the least significant 5 bits constitute a literal field which contains a small integer from 0 to 31.

Call Instruction

ALU Instruction

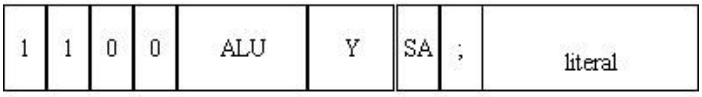


IF (Conditional Branch)

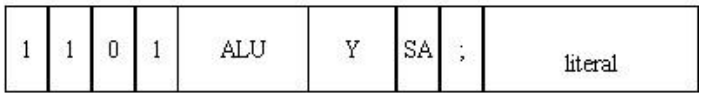
LOOP

ELSE (Unconditional Branch)

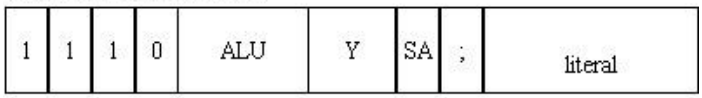
Literal Fetch Instruction



Literal Store Instruction



Memory Fetch Instruction



Memory Store Instruction

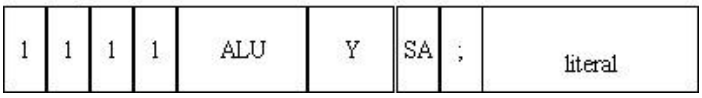


Figure 3.2. NC4000 Instruction Formats

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 bank number to be placed in the X-port to select one of 32 memory banks for memory access. In other instructions, this field is not needed and must be cleared to zero.

In a branch instruction, bits 12 and 13 determine the type of branch and the lower 12 bits supply the target address. The 12 bit address field specifies an absolute address within the current 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 4K word page boundary.

A 1 in this 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 drops T and jumps to the target address. Otherwise, IF is simply a DROP instruction. 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.

Another comment about this rather complicated instruction set 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 preclude some instructions or combinations of bits and these instructions should not be used. In Table 3.7-8, we have collected all the valid ALU instructions and instruction combinations that can be safely used in the NC4000P chip. Many of these restrictions will be removed when the production chip becomes available.

3.2. ALU Instructions

ALU instruction is the most complicated class of instructions in NC4000 chip because all 12 lower bits in the instructions are decoded to perform functions in parallel. A firm grasp on the use of the individual bits and their interaction is essential. Understanding will lead to an appreciation for the power of these instructions which can compress several Forth words into one machine instruction. A number of examples will also be given to illustrate how the field and bits can combine to form multiple function instructions. With this information 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 NC4000 can be of great help in explaining the inner mechanism of the ALU instructions, as shown in Figure. 3.3.

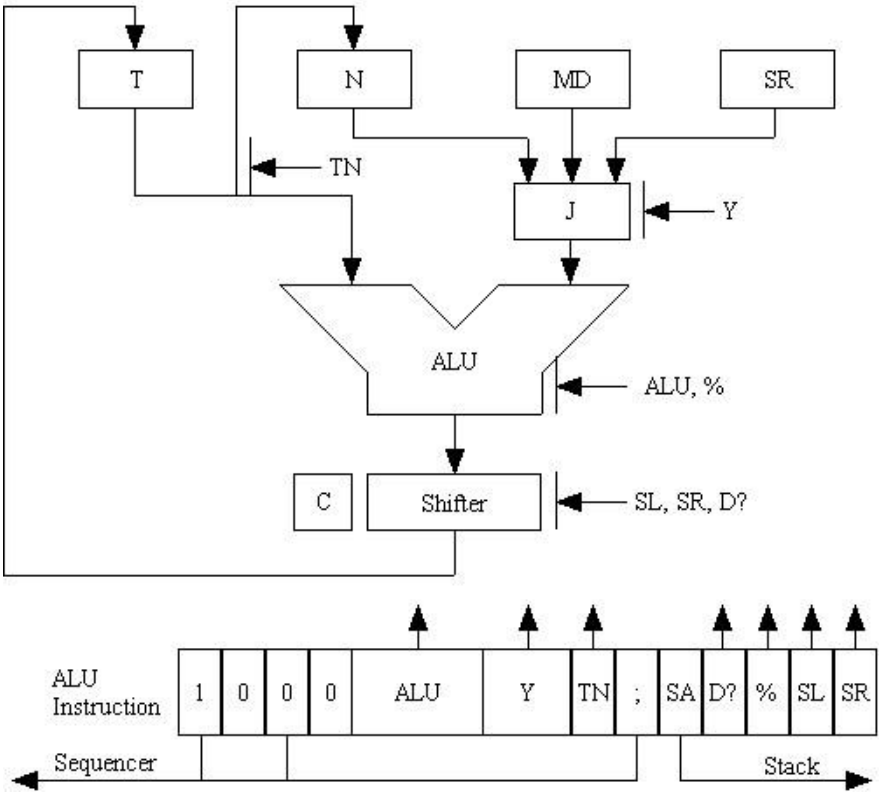


Figure 3.3. Data Paths and Registers in ALU Section

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, bits 9-11. 8 different actions can be performed on the two operands as shown in Table 3.1:

Table 3.1. 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

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 3.2.

Table 3.2. Y-Port Selector

Y Code	Source to Y Port
0	N register
1	N register with carry
2	MD Multiplier/divisor register

Normal operations use the N register as the Y-port operand. The N register with carry is selected when doing extended precision arithmetic. 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 connecting the N register to the output of T register. The content 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, a DUP operation is performed if TN is set and a NOP is performed if TN is 0. 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. Depending on the state of TN bit, the content of N register is pushed on the external stack or the top element on the external stack is popped into N register. The stack action can be summarized in Table 3.3.

Table 3.3. 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. Many other stack operations can be synthesized in conjunction with other activities in the ALU. The fact that these two bits are not in a contiguous field 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 the top of the return stack or the I register is popped into the Program Counter P and the Address Multiplexer A. The next instruction executed will be the instruction following the Call instruction and the execution 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.

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 3.4.

Table 3.4. 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, % 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 result 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 things can happen simultaneously, it is rather difficult to completely understand this ALU section and the different instructions the chip can perform. The best you 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 to compile the most compact machine instructions and to fully utilize the power of NC4000 chip. A better understanding of the inner mechanism of this ALU would enable you to anticipate the optimization process and thus assure production of the most efficient code.

Table 3.5. 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			

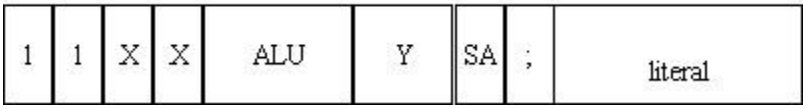
Special ALU Instructions

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

3.3. I/O and Memory Instructions

The I/O and memory instructions in NC4000 are characterized by one's in the two most significant bits and a literal value in the least significant 5 bits. 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:

Memory and I/O Instructions



However, there are many special cases causing the I/O and memory instructions to appear as if they are governed by random logic. According to Bob Murphy, who did the logic design of NC4000 with Chuck Moore, these instructions are controlled by random logic guided by logic equations. The best one can do is to present the entire table of valid I/O and memory instructions as shown in Table 3.6. Based on this table, we can make a few observations which might guide you in understanding this rather complicated instruction group.

The store bit, bit 12, will always control 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, the N register is used without carry.

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.

When bits 6 to 9 are all set (lxx7xx type), the instruction refers to the internal register specified by the 5 bit literal field. There are 17 addressable registers in the NC4000. Their register numbers, assigned names and functions are listed in Table 3.7.

Table 3.6. 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

Note: a-: SWAP , c-: SWAP -c

Table 3.7. 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 temporary storage.

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 to transfer data cross the I/O 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 functions of bits in these registers are shown in the following table:

Table 3.8. 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.

3.4. Graphic Models of Some NC4000 Instructions

The instruction set of NC4000 is quite overwhelming initially. There are too many bits in an instruction, and there are too many different combinations. The last two sections represent my best efforts in explaining this vast instruction set through words. In Volume 6 of *More on NC4000,* Timothy Huang contributed a paper on *Anatomy of the Forth Engine*, in which he

showed several drawings on how data flows in several NC4000 instructions. These drawings gave me the inspiration to present NC4000 instruction set in graphical form. As Confusius said: "A picture is worth more than a thousand words". Looking at the operations of NC4000 graphically might help many readers to understand this machine better. It was unfortunate that Timothy wrote that paper in Chinese, and not many NC4000 users could be persuaded to learn Chinese.

The drawings I developed so far deals only with the ALU instructions, which are the most complicated class of NC4000 instructions. They were presented in the NC4000 Users Group Meeting at Cupertino, California, January 23, 1988. Chuck Moore was also present at the meeting. His comment was that these drawings represent a good model of NC4000. This model only approximates the functioning of NC4000, because NC4000 was designed not based on this kind of models, but by a large set of logic equations compiled into gate array patterns. Many special functions in NC4000 are impossible to represent in simple graphical models.

3.4.1. Model of NC4000 ALU

Figure 3.4 shows a schematic drawing of NC4000 CPU which is slightly different from the drawings in Figures 2.5 and 3.3. This figure emphasizes the data flow paths and control paths in NC4000 around the ALU and immediately related registers. The data stack is also shown to illustrate its interaction with the N register. The N register is placed next to the shifter at the bottom of the ALU because the N register participates in shifting operations. The carry bit is also shown very prominently because it contains vital information, especially in extended precision arithmetic operations.

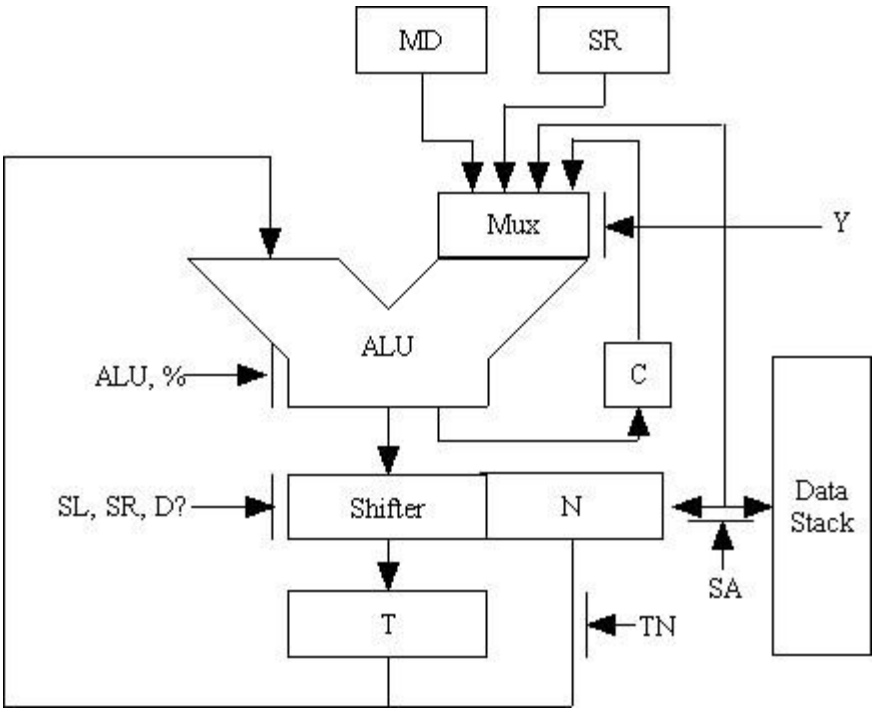


Figure 3.4. Another View of ALU

Bits and fields in the ALU instructions are shown to control some data paths and some logical elements. The control function cannot be shown completely because the bits and fields sometimes have more effects on NC4000 operations than simply opening or closing data paths. For example, TN and SA bits control the data path between T and N, and between N and the external stack.

Meanwhile, the TN bit also determines the direction of data flow between the N register and the external data stack.

The Y field controls the multiplexer at the right input of the ALU. The ALU field obviously controls the function of the ALU unit. The divide bit % is also sent to the ALU unit because it forces ALU to perform a conditional subtraction in the divide step and square root step instructions. The bits D?, SL, and SR controls the shifter and the N register to perform single or double integer shifts.

In most ALU instructions, the interaction between TN and SA bits produces the most interesting results, because of their effects on the data stack. The best way to analyze these effects is to further group the ALU instructions according to the bit patterns in the ALU field. For each group, TN and SA can then be assigned all possible combinations. The data paths and the end results can then be shown graphically.

In the following subsections, we shall discuss the following groups of ALU instructions: the SWAP group, the DUP group, the binary ALU group, the multiply/divide group, and the miscellaneous group.

3.4.2. The SWAP Group Instructions

The SWAP group contains the familiar SWAP, OVER, and DROP instructions. The fourth instruction DROP DUP is not a standard Forth operation. However, it discards the topmost item on the stack and duplicates the second item. The ALU selects the N register as its source through the Y multiplexer and passes its content directly into the T register without any modification. These four instructions are shown in Figure 3.5.

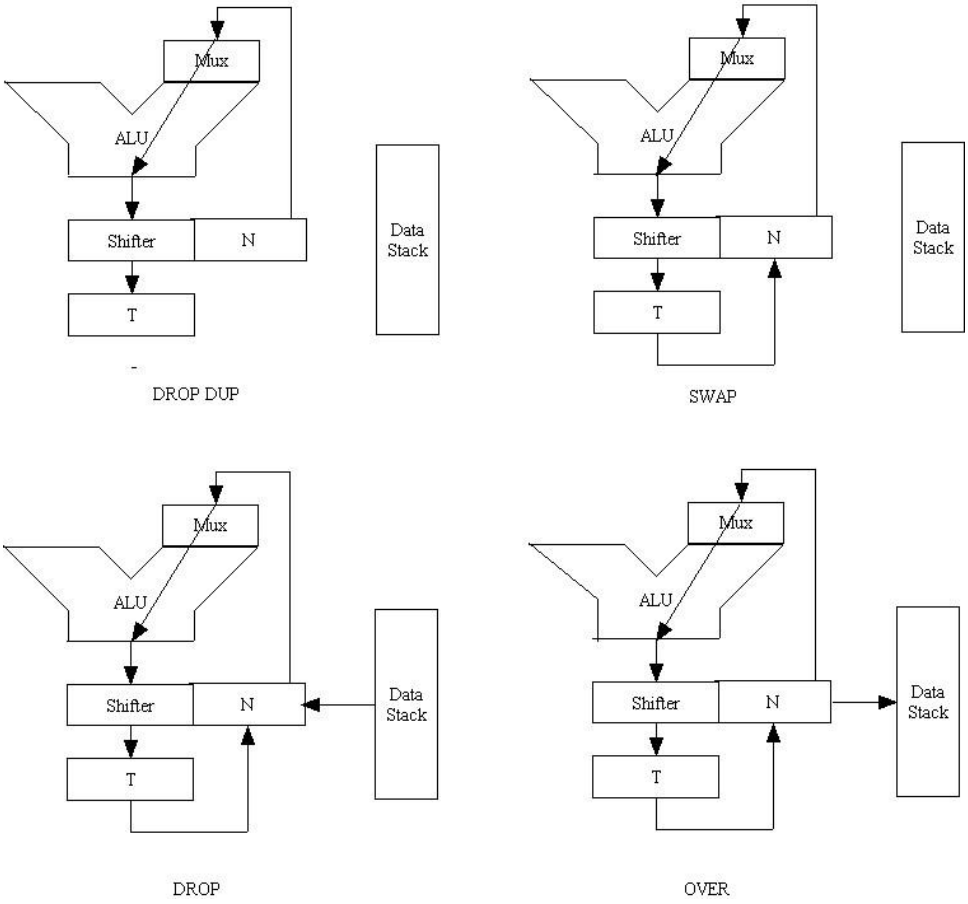


Figure 3.5. The SWAP Group

In the upper left is DROP DUP. In this instruction, both the TN bit and the SA bit are cleared. The external data stack is thus isolated from the N register, and the path from T register to N register is also broken. The content in N register is written into the T register and the original content in T is overwritten. The end results are that the top item on the data stack is DROP'ed and the second item is duplicated.

If the TN bit is set, the instruction becomes SWAP in the upper right corner of Figure 3.5. Here the content in T is copied into the N register while the original content in N is copied into the T register through the ALU. The top two items on the data stack is thus exchanged, which is what SWAP should do.

If the SA bit is set but the TN bit is cleared, you get DROP, as shown in the lower left corner of Figure 3.5. Since TN is off, the content in T is lost. The content in N is copied into the T register through the ALU. Because SA is set and TN is cleared, the top item on the external data stack is popped into the N register. The net effect is that the top of data stack, the original T, is dropped and all items under T are moved up one place.

When both TN and SA are set, the instruction is OVER. The original content in T is copied into the N register because TN bit is set. The original content in **N** is copied into T through ALU, and pushed

on to the external data stack, because SA and TN are both set. Hence two copies of the second item on the data stack are saved. The original top item is sandwiched between these two copies of the second item. OVER is thus synthesized in NC4000.

Every ALU function thus results in four NC4000 instructions with the permutations of TN and SA. They are all useful functions, though some of them may not have a standard Forth name.

3.4.3. The DUP Group

The DUP group is configured such that the ALU passes the content in T and stores it back into the T register. It is basically a NOP operation if both TN and SA are cleared, as shown in the upper left corner of Figure 3.6. Although NOP is not a standard Forth word, it is extremely useful and most real Forth system found it necessary to implement. The NOP instruction is especially important in NC4000; because when the data is passed from T to T, the shifter can be used to shift the data right or left by one bit before storing it back into the T register. These are the 2/ and 2* instructions which are implemented in most Forth system with machine code because they are used so frequently.

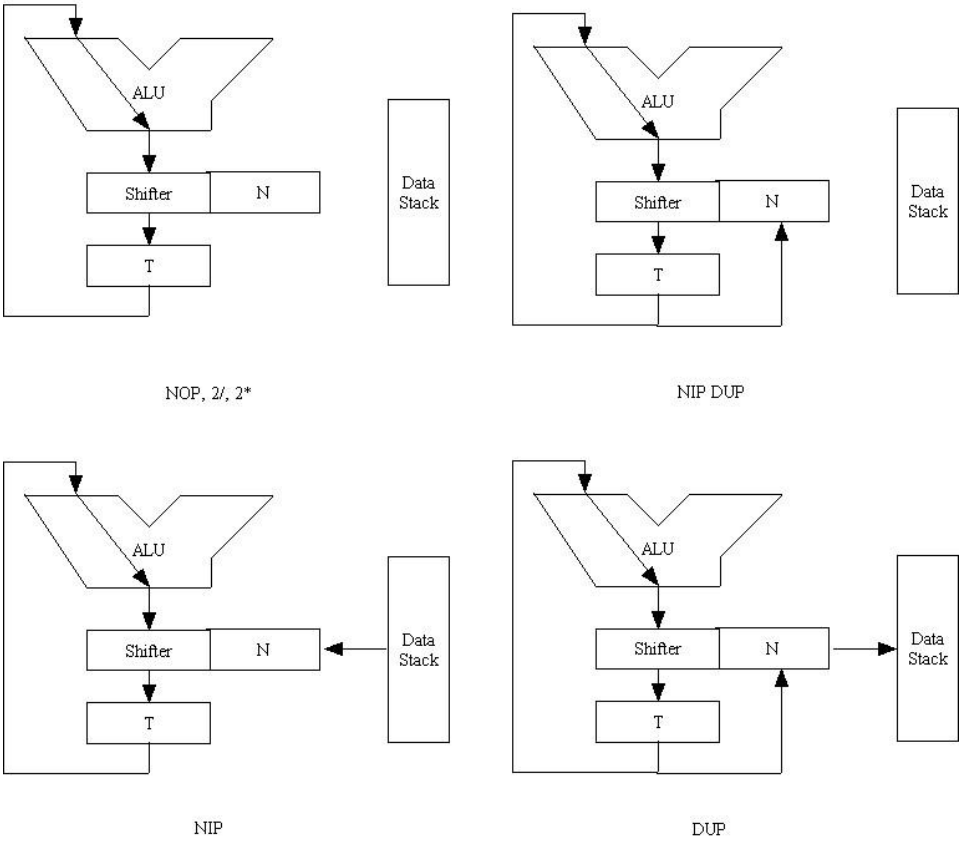


Figure 3.6. The DUP Group

If the TN bit is set, the content in T is saved both back into T and also into the N register. The original content of N is written over. Again, this instruction does not have a standard Forth name. It is equivalent to SWAP DROP DUP or NIP DUP.

If the TN bit is cleared and SA bit set, the result is SWAP DROP or NIP. NIP is not a standard Forth word but it is getting very popular lately. Most of the newer Forth systems have it. The T register is preserved through ALU. The N register, however, is overwritten by the top item on the external data stack because SA bit is set. As TN is cleared, the top element on the external data stack is popped into N and overwrites its original content.

In DUP, both TN and SA are set. Thus the T register is copied both into T and N, while the original content in N is pushed on to the external data stack. SA bit connects the N register with the external data stack, and TN bit specifies that data move from N to the external data stack.

3.4.4. The Binary ALU Group

Most of the arithmetic and logic operations in Forth pop the top two items off the data stack, perform some operations on them, and push the resulting item back on the data stack. This operation is achieved in NC4000 by the setting shown in the lower left corner of Figure 3.7. The ALU takes both T and N as its inputs and stores the results back into the T register. Since the TN bit is cleared, T will not be copied into N. Instead, the external data stack is popped into the N register because the SA bit is set and TN is cleared.

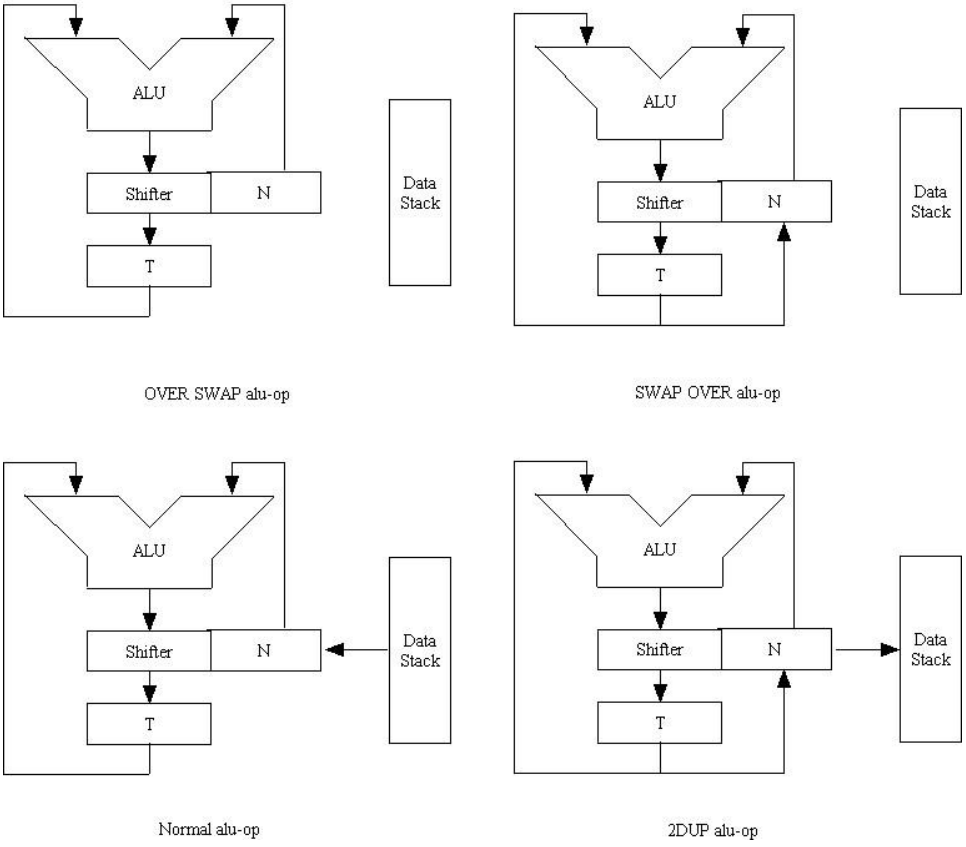


Figure 3.7. The Binary ALU Group

In this setting, the ALU can perform +, -, AND, OR, and XOR operations on the two input numbers from T and N. The subtraction in ALU can produce either T-N or N-T, depending upon the bit

pattern specified in the ALU field. Thus we have two subtractions: - and SWAP -. Both are single cycle instructions in NC4000. In both addition and subtraction, it is possible to include the carry bit generated in the prior arithmetic operation. The six possible arithmetic instructions are +,+ with carry, - , - with carry, SWAP- , and SWAP- with carry. They are sufficient to carry out extended precision mathematic operations.

One might add that the results of the arithmetic operation can be shifted right or left by one bit in the shifter before falling into the T register in the same machine cycle. The extra shift is very useful in many signal processing algorithms.

The normal binary ALU operation is only one among four possibilities. The other three possibilities are also quite useful because they preserve one or both of the input numbers, which are destroyed in the normal ALU operation. In the upper left corner of Figure 3.7, the content in N is preserved because both the TN and SA bits are cleared and the N register is isolated from T and the external data stack. This is equivalent to OVER SWAP alu-op, where the second item on the data stack is preserved.

If the TN is set and SA is cleared, as shown in the upper right corner of Figure 3.7, the content of the T register is saved by being copied into the N register. The results can be described as OVER SWAP alu-op in standard Forth terminology.

If both TN and SA are set, as shown in the lower right corner of Figure 3.7, the content in T is saved by copying into the N register. The original content of N is also saved by being pushed on the external data stack. The result is 2DUP alu-op or OVER OVER alu-op.

These three variants of the normal ALU instruction are useful because in many instances we have to duplicate arguments of an ALU operation so that they can be used later. These instructions complement the binary ALU operations in that one or both of the arguments can be saved.

3.4.5. The Multiply/Divide Group

16 bit multiplier is very expensive to implement in gate array technology, because it requires a large number of gates. In NC4000 design, Chuck Moore chose the most economical solution: only a small set of logic is needed to perform conditional addition and subtraction, which could then be repeated to accomplish 16 bit multiplication and division. By adding an extra register, he could even do the square root by repeating the conditional subtraction. According to his design, multiplying two 16 bit numbers to form a 32 bit product needs only 16 machine cycles. To divide a 32 bit number by a 16 bit divisor to form a 16 bit quotient and a 16 bit remainder needs only 18 machine cycles. To take the square root of a 32 bit number also needs only 18 cycles. By modularization the logic, these rather complicated operations can be realized in NC4000 with fewer gates and higher speed than the designs in conventional microprocessors like 680x0 and 80x8x.

Figure 3.8 shows four of these complicated operations. The fundamental operation is the D2* and D2/ operations shown in the upper left corner of Figure 3.8. The ALU is set up to pass T through without modification. By setting the D? bit in the instruction, the shifter at the bottom of ALU is now combined with the N register to form a 32 bit double integer shifter. If SL bit is also set, the 32 bit

integer in the T-N register pair is then multiplied by 2. If SR bit is set, the T-N register pair is then divided by 2. Multiplication steps, division steps, and square root step all use this double integer shifter to adjust the product.

The multiply step instruction *', octal 104411, can be decoded as follows: adding MD register to T, then shifting the results right with the N register by one bit. The assumptions are that MD register holds the multiplicand, the N register is initialized to the multiplier, and the T-N register pair holds the partial product. Simple addition cannot generate the product. The trick is to use additional logic to detect the least significant bit shifted out of the N register. If this bit is a one, the sum of T and MD is shifted and stored into the T register. However, if the bit shifted out of the N register is a zero, the addition is suppressed and T is passed unmodified into the shifter. This conditional addition controlled by the least significant bit in N achieves one step of multiplication. By repeating *' 16 times, a 32 bit product is produced in the T-N register pair.

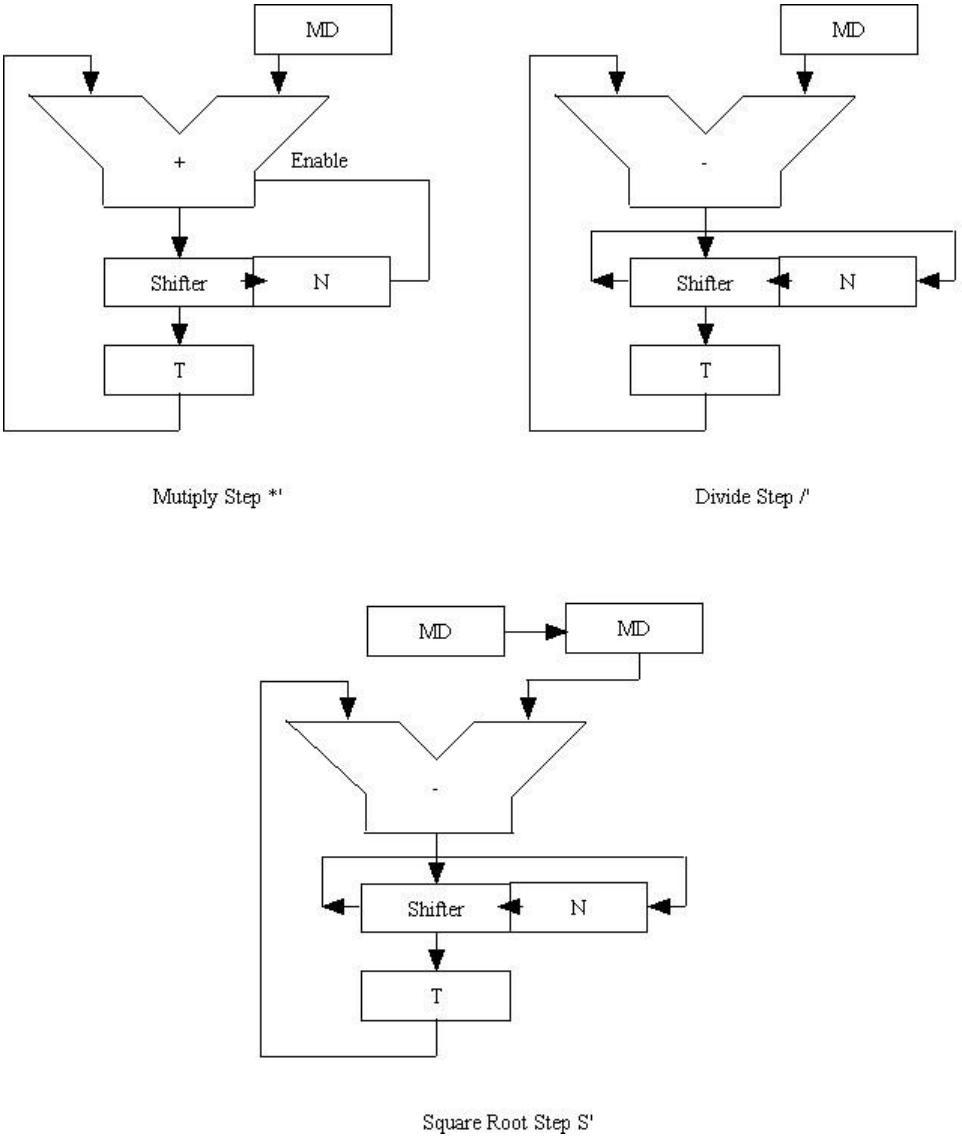


Figure 3.8. The Multiply/Divide Group

The infamous multiplication bug of NC4000P, that it cannot multiply odd numbers, is the result of a mistake which ignores the least significant bit in the MD register during addition. Thus if the multiplicand in MD is odd, the least significant bit is ignored.

Repeating *' 16 times produces a 32 bit unsigned product from an integer multiplicand and an unsigned multiplier. If the multiplier is a signed, 2's complement number, the resulting product would be erroneous. For a signed multiplier, the last step of the multiplication should use the signed multiply step *instruction, octal 102411. This last step does a conditional subtraction instead of a conditional addition (the ALU code is 2 instead of 4). This step correctly interprets the most significant bit in the multiplier as a sign bit instead of as a regular bit 15. The result will be a 32 bit 2's complement product.

NC4000 provides another interesting multiply step, the fraction multiply step *F instructions, octal 102412. This step should also be applied as the last step in multiplication. It is similar to *- in that a conditional subtraction is performed. However, instead of shifting the product to the right, it shifts the product to the left by one bit. The reason is that the multiplier and the multiplicand are now interpreted as fraction numbers; i.e., the most significant bit is still the sign bit, but the next bit is the first bit after a binary point placed between bit 15 and bit 14. If a 32 bit product is produced the normal way, the binary point should be placed between bit 29 and 28. In the last step, *F shifts the product to the left, and thus restores the binary point between bit 31 and 30, forming the correct fraction product.

The divide step /' instruction, octal 102416, is shown in the lower left corner of Figure 3.7. The bits in this instruction can be interpreted as: subtract MD from T and shift the results with the N register left by one bit. This does not sound like a division. The trick lies with the % bit, which is set in this instruction. This % bit controls the ALU to effect a conditional subtraction. If the subtraction is successful and no carry is generated from the subtraction or T>=MD, then the difference is sent into the shifter and shifted with N. If a carry is generated in subtraction, meaning that MD is greater than T, the subtraction is nullified and the unmodified T is sent to the shifter. At the same time, if subtraction is successful, a one is shifted into the least significant bit of N; otherwise, a zero is shifted into N.

To begin the division process, the 32 bit dividend is placed in the T-N register pair and the 16 bit divisor is put in the MD register. The divide step /' is repeated 15 times. The last step of division requires another divide step, the last divide step /", octal 102414. /" does the same thing as /', except that the results of the conditional subtraction are not shifted, since the SL and SR bits are cleared in this instruction. The reason is that the remainder of the last divide step must not be shifted, or the remainder would be only half of the correct value. The division is restricted to positive dividend and unsigned divisor. The quotient will be in N register and the remainder in T register.

The square root step S', octal 102616, is shown in the lower right corner of Figure 3.7. It literally means that the SR register is conditionally subtracted from the T register, and the result is shifted left with the N register. Square rooting is of course much more complicated than that. Logically, both the MD and SR registers are involved in a subtle fashion to achieve the desired goal. At any step, the T-N register contains the current remainder of the square root, the MD register holds the partial root, and

the SR register holds the least significant test bit. The whole thing is quite similar to the situation when you do long hand square rooting. The partial root in MD is OR'ed with the test bit in SR and subtracted from the T register. If T is less than this test number, the subtraction is nullified and T is passed into the shifter. If T is greater or equal to the test number, the subtraction results is passed into the shifter. The shifter and N is shifted left by one bit. If the subtraction is performed, a one is also shifted into the N and MD registers. The test bit in the SR register is shifter right by one bit. As the T-N pair is shifted to the left and the SR register is shifted to the left after a S' step, the remainder in T-N and the root in MD-SR are displaced by 2 bits as required by square rooting.

To start this square root process, the 32 bit squared value is placed in the T-N register pair, MD must be cleared to zero, and SR is initialized to 32768, which is the initial test bit pattern. S' can then be repeated 16 times. The square root is generated in both MD and N, and T contains the remainder. The restriction is that carry cannot be generated in any step during this process. This restricts the root to be less than 16192 in the prototype NC4000P version.

These more complicated mathematic operations show that the model we are using to explain the function of NC4000 is not quite adequate; multitude of additional logic exists in NC4000 to enable it to perform such diverse tasks in a small package. Nevertheless, this model of NC4000 ALU is very useful in capturing the major activities of NC4000 in these various instructions.

3.4.6. Miscellaneous Instructions

There are many other interesting instructions in NC4000 other than the ALU instructions. They are in the general category of 10 and Memory instructions. It is interesting to relate function with the bit patterns in these instructions, especially when you have to decipher octal dump of a code segment. As mentioned in the last section, bits 6-8 in the IO/Memory instructions do not follow fixed patterns and thus are very difficult to interpret. Thus no attempt will be made here to explain the bit pattern in this field.

The return stack is not easily manipulated by the user. However, the top most element of the return stack is cached on chip as the I register. This register can be accessed by many instructions using the register 10 instructions. The I register is assigned two register numbers, 1 and 17. Normally when the I register is used as the top of the return stack, it is addressed as register 1. If it is used as a count register, it is addressed as register 17.

The instructions >R, octal 157201, and R>, octal 147321, are of special interests as shown in Figure 3.8. The data stack and the return stack in NC4000 can be viewed as a 515 cell register array, with the I, T, and N registers at the center. The entire array can be shifted to the left by >R and to the right by R>. The three registers at the center of this large array is a window by which ALU has access to the array.

Three useful instructions related to the return stack through the I registers are: R@ or I, octal 147301, which copies the content in I and pushed them into the T register; R> DROP, octal 140721, which discard the I register and pops the external return stack; and R> SWAP >R, octal 157701. The last instruction looks very complicated. It is actually very simply because it is a register exchange instruction I@!, by which the content in T is exchanged with the content of another on-

chip register, which happens in this case to be the I register.

The TIMES instruction, octal 157221, is very similar to >R. As the content in T is pushed into the I registers, NC4000 simultaneously latches the next instruction in the Instruction Latch register L and repeats this instruction until I register is decremented to zero. Extra logic in NC4000 decodes this instruction to accomplish this single instruction repeating function.

All the memory fetch instructions, such as the regular @, local memory fetch, long and short literal fetch, and internal register fetch, get data from memory and deposit the data into the T register. The data is fetched through the Memory Port register M and passed into T through N and ALU. Therefore, some arithmetic or logic operation can be performed on the data fetched before storing it into the T register. As memory fetch usually takes two machine cycles, the second cycle can be used to do one more ALU operation on the data and save a cycle.

In accessing large arrays of data stored in memory, the T register usually holds the address pointing to the array. Normal fetch instruction destroys this address, making it *very* awkward to program because addresses have to be saved and restored to the T register. NC4000 provides two powerful instructions to help these array operations: @+ (1647nn), @- (1627nn), !+ (1747nn~, and !- (1727nn). These instructions retain the array address in the T register. Moreover, they allow a small number between 0 and 31 to be added to or subtracted from the address in T simultaneously with the memory operation. Using these instructions with the TIMES repeating instruction, you can scan very large arrays in NC4000 using only one machine cycle per access, because the memory address is generated automatically and the memory access instruction is latched in L. Some of these array operations are shown in Figure 3.9 also.

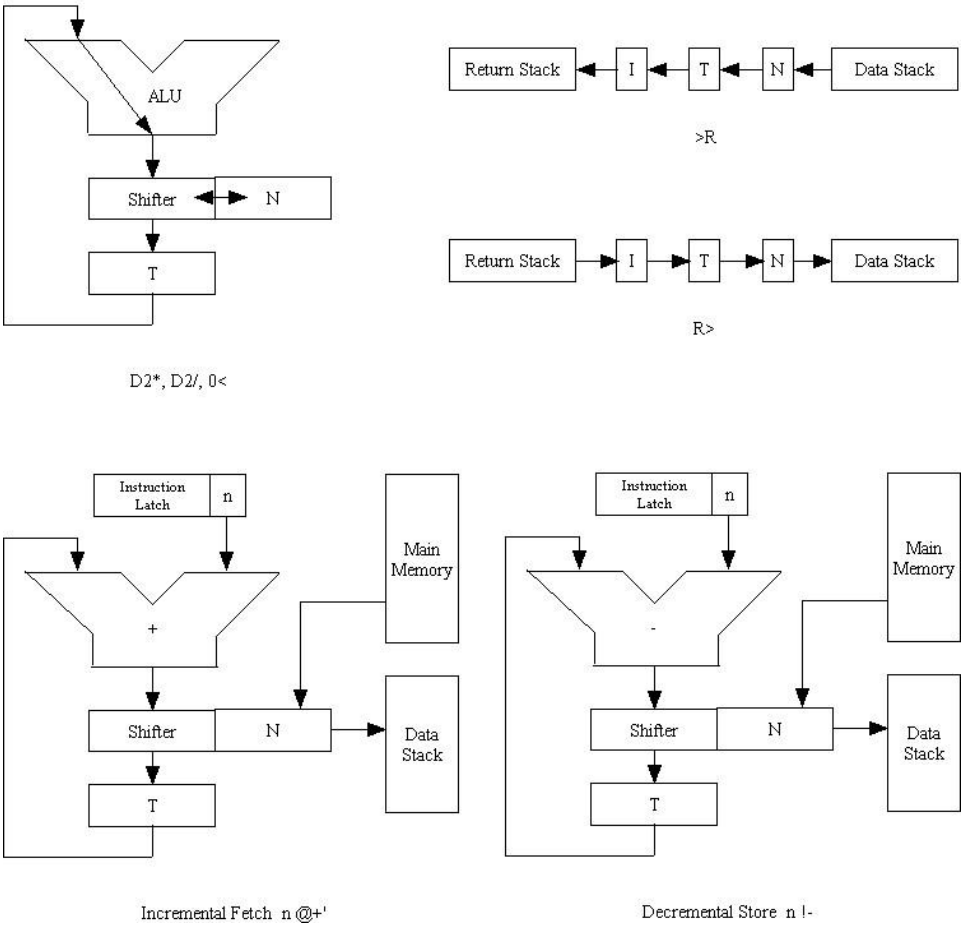


Figure 3.9. Return Stack and Array Operations

Chapter 4. NC4000 Computers

4.1. Commercial Products Using NC4000 Chip

One of the major design goals Chuck Moore wanted to achieve with 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 NC4000 chip. Many others have been custom built to solve specific problems.

4.1.1. Early Alphabetic Boards

When the first wafer of NC4000 was diced at Mostek in early 1985, 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" by4" 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" by14". It used high speed RAM's and ROM's and was specified to run at 7-8 MHz. 56K 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 pins were reduced from 128 to 121. Novix modified the Beta Boards to accommodate the new NC4000 chip. Novix also incorporated the new Beta Boards into a package with hard disk drive and floppy disk drive for software and hardware developers.

Chuck Moore updated the design of Alpha Board for the new NC4000 chips and distributed it as a Gamma Board kit, which consists of a NC4000 chip, a 6" by 4" PC board, and a pair of 2732's with cmForth firmware. This kit was sold through his company Computer Cowboys in Woodside, California. The kit also contains 360 Augat Holtite 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 resisters 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, also built a single board computer based

on NC4000 chip. This single board computer was called Delta Board with a code name SC1000. It was 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 and in kits. Software Composers is also producing supporting accessories such as power supply, back plane, expansion memory board, etc.

These alphabetic boards represented the early efforts to incorporate NC4000 chips into usable computer designs. Since NC4000 is very easy to use and very forgiving in the power supply and interfacing requirements, many people used it to build their own systems for very specific applications. It is impossible to document all these computers.

4.1.2. ForthKits from Computer Cowboys

Computer Cowboys is Chuck Moore's company selling NC4000 products and services. The Gamma Board was its first NC4000 product, which was officially named as ForthKit 1 or Fk1. Since Chuck owns this company, he is freed from all the customary constrains in developing new products. As demonstrated in the design of the ForthKit, he showed no respect to common engineering practices. The chips are mounted on both sides of the PC board to minimize the board area, which terrified many electronic engineers. Memory and I/O buses are also brought out on both sides of the PC board.

Chuck built boards in batches, typically 30 to 50 boards in a batch. After these boards were sold out, his would design a new board, incorporating all the new ideas developed with the old board. Hence we have seen the ForthKit 2, ForthKit 3, and now he is at ForthKit 4, as of late 1987.

Improvements in the ForthKit 4 over the previous versions include the following:

- Power and I/O routed to 3x32 100 mil header (DIN 41612)
- Main memory, two stacks, and I/O spaces each having their own 2x20 headers for expansion
- Provision for MAX232 to generate true RS-232 signals
- Floppy disk interface, and video display interface.

The ForthKit 4 is priced at $503.00, including NC4016 CPU, Fk4 board, 10 MHz clock, cmForth in PROMs, RAMs, Augat Holtite sockets, and miscellaneous parts. Assembly and testing is $100.00.

4.1.3. Products from Novix, Inc.

Novix is still the only source of NC4000 chips. The NC4000P prototype chip was renumbered as NC4016. However, the chip is exactly the same as NC4000P, with all the known bugs. Novix had announced the much improved version, NC6000/NC5000 series of chips. Samples of NC6000 were delivered to Novix by Mostek in late 1987. It will be available in early 1988.

Following is the Novix product list:

Beta Board: 64K words memory, 2 serial ports, SCSI interface, polyForth OS $3500.00

Micro Mainframe: Beta Board, polyForth, floppy drive, Winchester drive, optional tape backup, $7500.00

NB4100 PC Coprocessor Board: 128K high speed RAM, PCDOS interface, 4K byte dual port RAM bus interface, $1250.00

NB4300 STD Bus Card: 128K bytes high speed RAM, one RS-232 port, 20 bit address and 8 bit bus connection, $1170.00

Tiny Turbo 4000 Board: 4K PROM, 8K CMOS RAM, 4.5x6.5" board, Novix 83-Forth, $595.00

NS4100 Small C Compiler $149.00

4.1.4. Products from Silicon Composers

Silicon Composers, originally the Software Composers, was formed by Dr. George Nicol to produce NC4000 based computers. It started with the Delta Board and has developed a range of products using NC4000 as the CPU. Recently, it is marketing the FORCE chip set from Harris Semiconductors on an IBM AT coprocessor board, which is the first commercial product sporting the Harris Forth engine. Its product list is as follows:

Delta Board SC-1000CPU: 4K ROM, 8K RAM, one RS-232 port, SC-1000 2x36 edge connector bus, $795.00

Delta Evaluation System: Delta Board with power supply SC-1000DES and RS232 cable, $895.00

Delta Development System: Delta Board, 7 slot backplane, Model 1 56K words CMOS RAM, power/battery card, $1495.00

PC4000 Plug-in PC Board: IBM PC coprocessor board, 512K byte RAM, PC bus interface, SCForth development language, $1495.00

AT/FORCE Coprocessor Board: Harris FORCE core chip set, 7 MHz clock, 32K bytes RAM, AT bus interface, optimizing compiler/ linker FCOMPILER, $4500.00

A range of software products supporting the above hardware.

4.1.5. Other Companies and Products

FB-4016 from Forth, Inc.: IBM PC coprocessor board, 128K bytes RAM, 16K bytes dual ported RAM for DMA with PC, polyForth operating system, up to 6 boards in a PC, $3450.00

V4000 CPU Board from VME Inc.: standard VME bus, 128K bytes dual ported RAM, two RS-232 ports and a parallel port on P2, $4500.00

Novix Personal Computer byNovix Solutions: ForthKit 3 board, 32K words RAM, BOOK bytes 3.5" floppy drive, modified cmForth, $1000.00

BASEBOARD by SoCal Skunkworks: 16K to 128K of RAM, true RS-232 port, reverse power protection, SC-1000 bus connector, $400.00 for board with 128K RAM.

4.1.6. List of Manufacturers

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

Forth, Inc.
111 N. Sepulveda Blvd., Manhattan Beach, CA 90266
(213) 372-8493 (inside California) (800) 55-Forth (outside California)

Novix Inc.
19925 Stevens Creek Blvd., Suite 280 Cupertino, CA 95014 (408) 255-2750

Novix Solutions
7067 Mayhews Landing Road Newark, CA 94560 (415) 796-1037

Silicon Composers
210 California Ave., Suite I Palo Alto, CA 94306 (415) 322-8763

SoCal Skunkworks
5358 East Falls View Drive San Diego, CA 92115 (619) 583-5730

VME Inc.
560 Valley Way Milpitas, CA 95035 (408) 946-3833

4.2. Build Your Own NC4000 Computer

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

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

4.2.1. The CPU Section

NC4000 chip, its control and I/O connections are shown in Figure 4.1. 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 NC4000 chip.

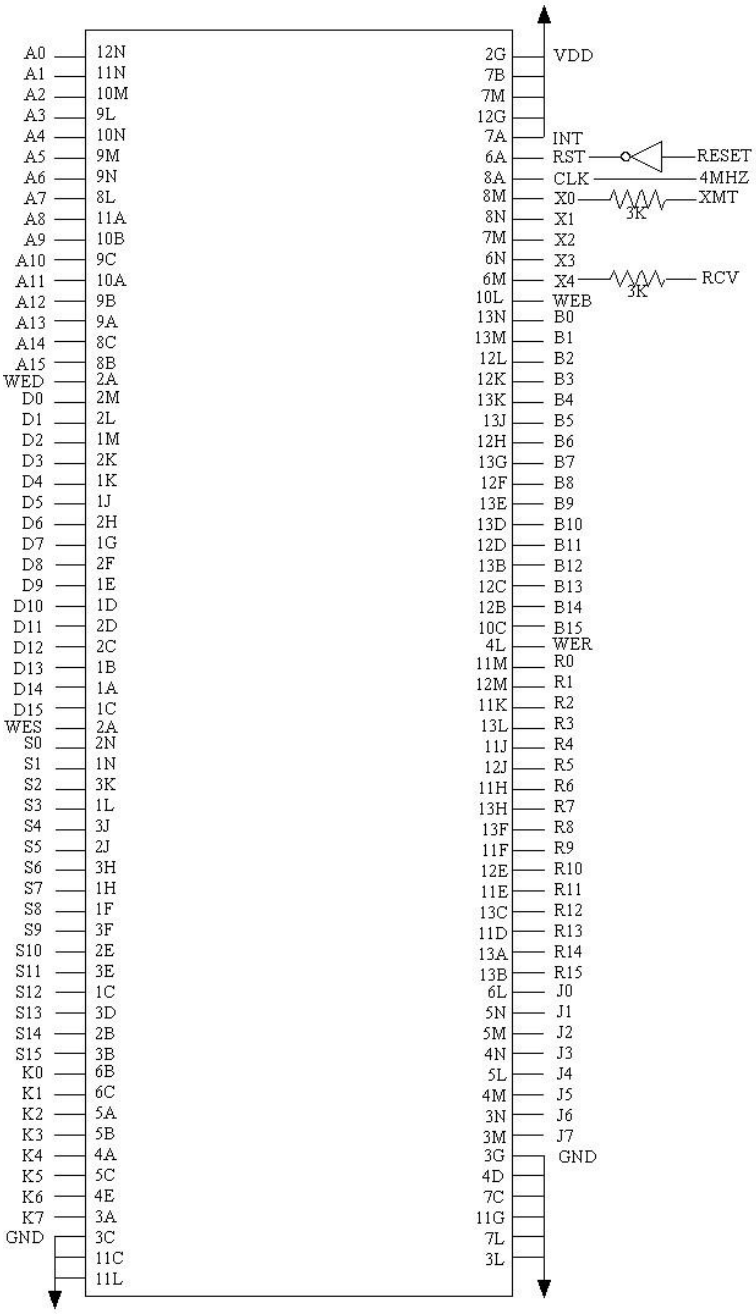


Figure 4.1. CPU Section of a NC4000 Computer

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 you 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, all 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.

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. To bring the RST pin low, the driving chip must be able to sink 60 mA of current. 74HC132 or equivalent is required.

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 20H 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 20H 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 S(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.

4.2.2. 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 un-terminated inputs.

Each of the output lines thus configured can draw 60 mA from the device connected to it. If you are going to connect other devices to these ports, be sure that the devices 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 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. Two resistors are needed to prevent damages to NC4000 because the RS232 device may have the transmitter and the receiver pins reversed.

4.2.3. Main Memory

In the design of a small computer with NC4000 as the central processing unit, there are two important constrains 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 1FH 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 easy 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 Figure 4.2. 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 OFFFH and from 2000H to 2FFFH, while the PROM's reside between 1000H and 1FFFH.

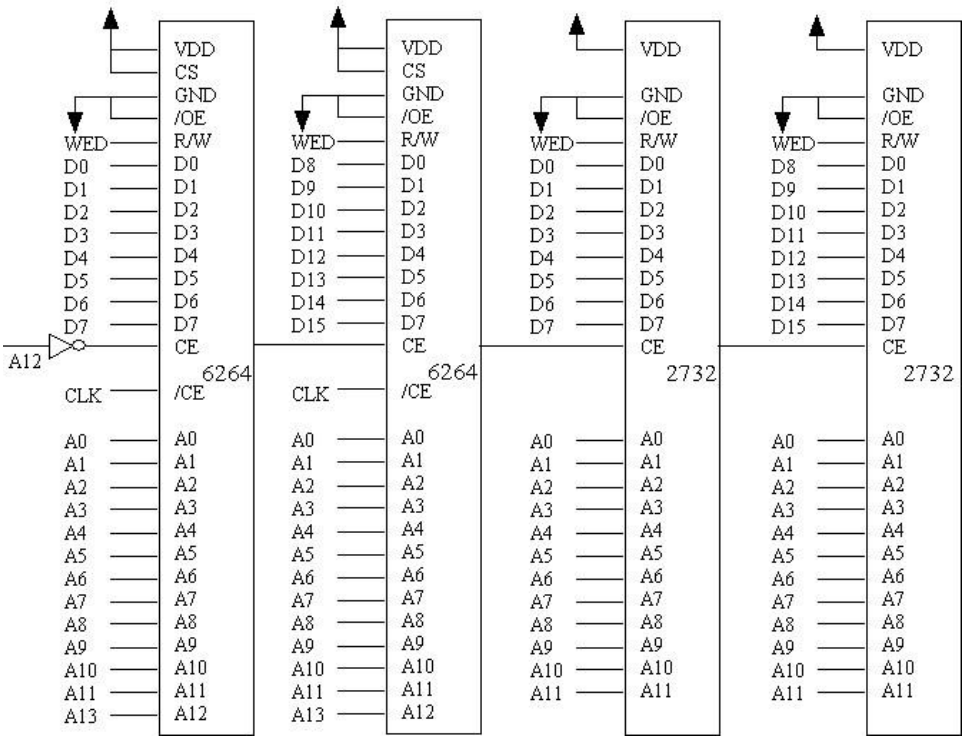


Figure 4.2. Memory Decoding for a Small NC4000 Computer

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 Figure 4.3 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 useful 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.

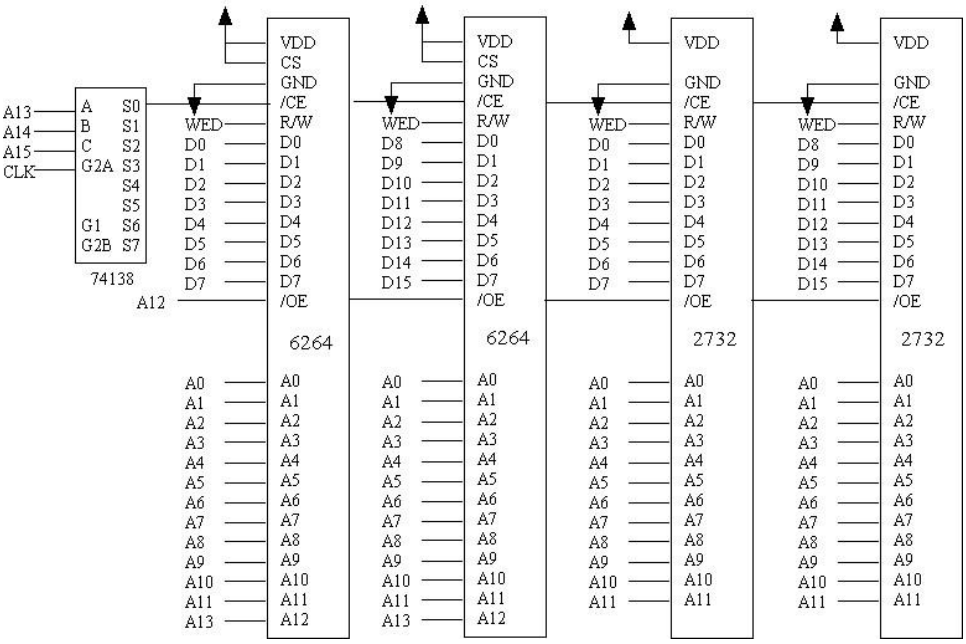


Figure 4.3. Memory Decoding of a Large NC4000 Computer

4.2.4. Data Stack and 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 Figure 4.4 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.

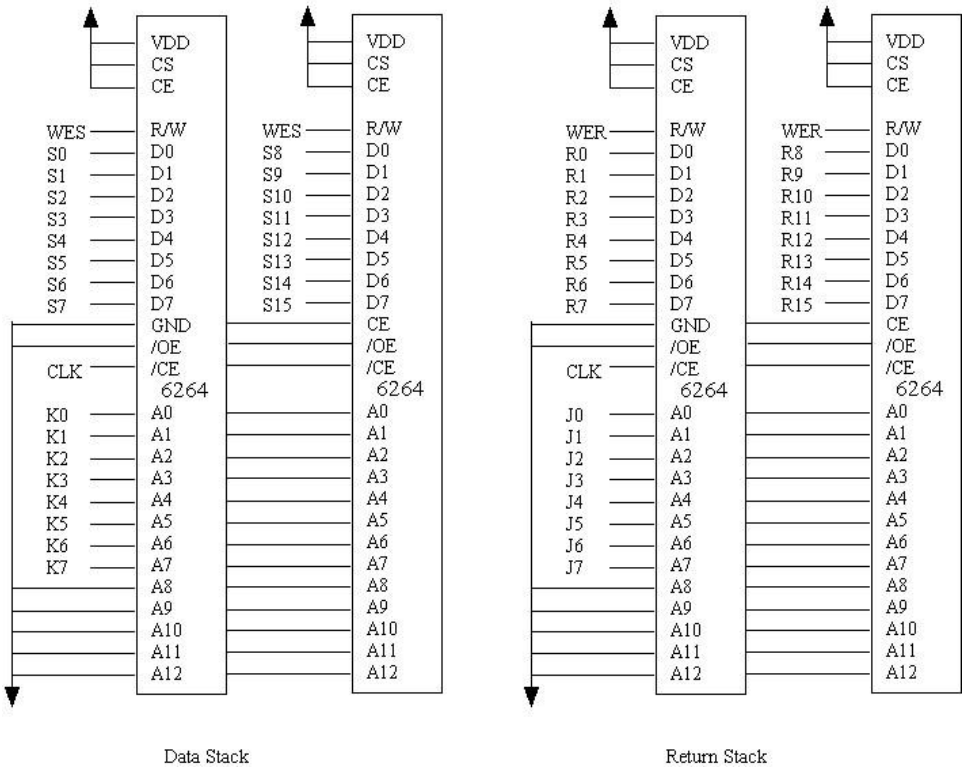


Figure 4.4. Data and Return Stacks of a NC4000 Computer

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.

4.3. 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" by4" PC board is more than enough to host these chips. For those who can handle wire wrap 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 Silicon Composers are supplying PC boards with NC4000 chip, it is worth the extra cost to buy their kits and do the assembly yourself.

A final note on the power supply. 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.

4.4. Hardware Enhancements

In designing NC4000, Chuck Moore wanted it to run as fast as it possibly can, and the speed of program execution is the primary consideration. To realize a complete CPU with only 4000 gates, the design must be very efficient in gate count. To achieve the highest speed with limited number of usable gates, many desirable features had to be sacrificed. The trade-off between functions and speed is particularly apparent in that the top half of the memory space cannot be used to store executable code, and that branching and looping are limited to absolute addresses within the current 4K word segment. The very heavy emphasis on the B port and X port is also indicative of Chuck Moore's intention to use NC4000 as high speed controllers rather than general purpose computers. Limiting the external stacks to a depth of 256 elements is acceptable for most applications, but not adequate to support recursions in general.

Because of the large number of signals brought out to the pins, most of the limitations in NC4000 can be eliminated with appropriate external hardware. In this section, we shall explore some possibilities to circumvent these limitations and enhance the functionality of NC4000.

4.4.1. PAL Memory Decoder OF5138

The architecture of NC4000 makes it very difficult to optimize the memory design, especially for systems which requires large amount of memory. There are two constrains in configuring memory for NC4000. The reset vector must be located at 1000H, requiring that ROM memory must be at this location. The local memory from 0 to 1FH is optimized to store frequently accessed variables because it can be addressed by single word instructions. Thus we need RAM memory from 0 up. In the current designs of NC4000 computers, memory is generally decoded in 4K word pages so that RAM can be assigned to addresses 0 to FFFH and ROM to 1000H to 1FFFH. As large size memory chips become cheaper and readily available, decoding in 4K word pages becomes inefficient and often awkward for deriving a sensible memory design.

Table 4.1. Pins of OF5138

Pins	Function
Vcc	5 V power supply, +-10%.
Vdd	Ground and power return.
A12-A15	Input. Connect to address lines from NC4000.
X1-X3	Input. Connect to extension port of NC4000 to select memory banks.
/CLK	Input. Connect to the master clock driving NC4000 system. Frequency limit 4 MHz. Low period gates the bank select signals to output pins.
/EN	Input. Low active chip enable.
EN	Input. High active chip enable.
ROMO	Output. Low active. Select ROM memory from 8000H to FFFFH. It is also active for memory between 1000H and 1FFFH where Forth kernel must reside.
RAMO	Output. Low active. Select RAM memory from 0 to 7FFFH, except the region between 1000H and 1FFFH.
RAM1-RAM6	Output. Low active. Select one of 6 banks of RAM or ROM memory to

respond to addresses from 8000H to FFFFH, in place of ROMO bank.

OF5138 is a PAL chip I designed specifically for large NC4000 computer systems. With this single chip decoder, one can build a wide range of products using as little as 8K bytes each of ROM and RAM memories, or as much as 512K bytes of ROM/RAM memories for memory intensive applications.

The optimal size of ROM or RAM chips for NC4000 is 32K bytes because NC4000 can only address 32K words as program memory. The 32K byte PROM chip, 27256, has been available in quantity for more than two years now. The static RAM chip of the same size is now available from Toshiba as 43256. The price of 43256 is expected to drop to $10 within 1987. OF5138 can decode banks of these 32K byte memory chips directly. If one uses smaller memory chips, additional logic is necessary to select chips inside 32K word banks.

OF5138 is housed in a 20 pin DIP plastic package. The pin-out is shown in Figure 4.5. The function of each pin is summarized in Table 4.1.

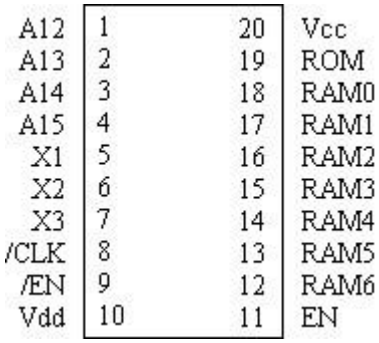


Figure 4.5. Pinout of OF5138 Decoder Chip

Figure 4.6 shows a memory system which can be decoded bya single OF5138 decoder chip. It includes a RAM memory bank in the memory range between 0 and 7FFFH, a ROM memory bank between 8000H and FFFFH. The ROM memory bank shares its address space with the other 6 banks of RAM/ROM memory. The RAM memory banks can accommodate static RAM chips of sizes from 4K bytes to 32K bytes. The largest RAM chip is 43256. The ROM chips can also range from 4K bytes to 32K bytes (2732 to 27256).

A minimal system would use two 2732's to store the Forth kernel and two 6116's for the kernel to operate. Note that RAM memory between 1000H and 1FFFH cannot be addresses because the decoder would activate the ROM memory and use code stored in the physical memory between 9000H and 9FFFH. This range of ROM is always activated for code execution no matter which bank of the upper memory is selected to ensure that the Forth kernel is always available to NC4000.

The maximum memory OF5138 can handle is 512K bytes divided into 8 banks of 64K bytes each. The most convenient implementation would use 27256 for ROM's and 43256 for RAM's. If one desires even larger memory space for memory intensive applications, it is possible to use additional X-port or B-port pins in NC4000 to drive the /EN and EN chip enable inputs. This method allows

the designer to use several OF5138 decoders to select more than 8 banks of memory.

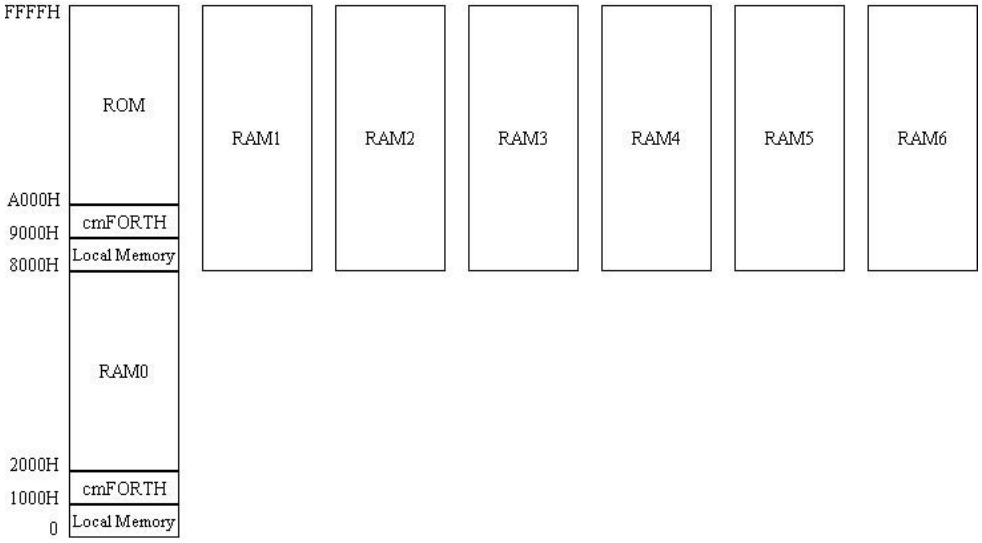


Figure 4.6. Memory Map of a 512K Byte System

4.4.2. Stack Expansion Counter OF5493

OF5493 is an 8 bit up-down counter chip specifically designed for NC4000 Forth microprocessor to expand its data stack to 64K words, making the stack space a useful buffer area to store large amount of data transferred between the memory space and the I/O space. The output bits from the counter become the higher byte of the stack pointer register, extending the stack pointer (K register) from 8 bits to 16 bits, as shown in Figure 4.7.

OF5493 contains an 8 bit up-down synchronous counter with a common clock to load the counts. The counter is incremented whenever it detects a transition from all high to all low in the 8 input pins AO-7. The counter is decremented when the input pins make a transition from all low to all high. This behavior makes the 8 counter output pins act like an 8 bit extension of AO-7 used as a stack pointer. Thus if we connect A0-7 to the data stack pointer output K0-7 from NC4000 chip, and /CLK to the inverted master clock in NC4000 system, the combined A0-15 lines provide a 16 bit stack pointer to address the static memory used as external data stack. The data stack is thus extended from 256 words to 65536 words if all the address lines are utilized.

As NC4000 provides valid stack pointer K0-7 at the trailing edge (high to low) of the master clock CLK, and the OF5493 counter is latched by the leading edge (low to high) of the clock, CLK must be inverted before connected to the /CLK input. Internally, the setup time between when addresses are valid and the latching of the counter should be $_{35}$ ns nominal, to allow the input signal to propagate through the gates to the output latches. This condition is generally satisfied when the clock frequency is 4 MHz or lower, because the stack pointer address is valid 60 ns ahead of the trailing edge of the master clock.

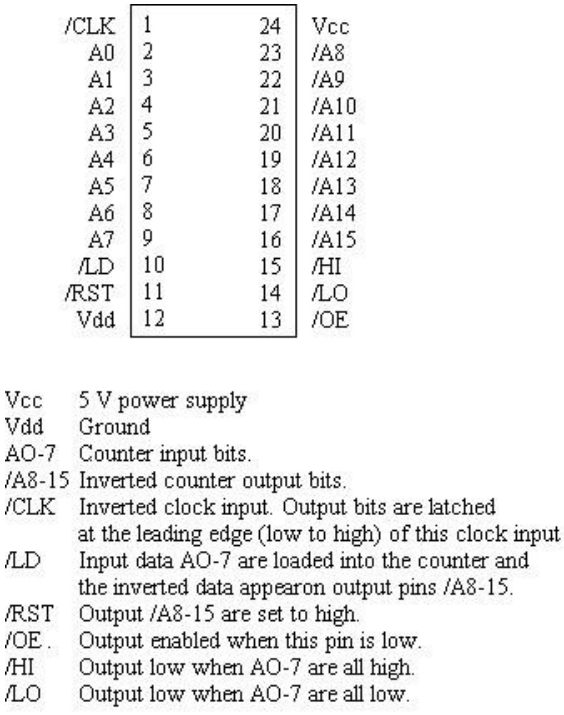


Figure 4.7. Pinout of OF5493 Counter Chip

The counter can be preset to all high output bypulling the /RST line low, orit can beset tothe input AO-7 bypulling the /LD line low. These control signals are useful toset the counter to a known state. However, if the counter is used only as an extension to the data stack pointer, it does not have to be set toany specific value. Thus the /LD and /RST lines can be left open ortied toVcc for NC4000 application.

Figure 4.8 shows how OF5493 can beused toextend the data stack of NC4000 from 256 words to 65536 words by expanding the data stack pointer. Note that an extra inverter gate is needed toinvert the clock from NC4000 tolatch the counter. The same circuit can be used for the return stack. The return stack needs to be extended if NC4000 will be used in applications which require often recursion.

4.4.3. Another Novel Memory Decoding Technique

This novel memory decoding method was first developed byRick Van Norman. It was used in his Novix Solutions Computers and was lately adopted byChuck Moore in the ForthKit 4.

The basic premise of this method is that 256K bit static RAM chips are becoming cheaper and more available. It is more economical touse them than to use the older 64K SRAM or smaller for program memory in NC4000 systems. Two 256K SRAM fit comfortably in the lower 32K words of NC4000 memory map. Since we need some boot-up PROMs at 1000H to host a Forth kernel, we will put a pair of PROM's at this location for boot-up only. Once the system is up, the kernel will copy itself into SRAM and let the PROM's go tosleep. After this copying process, the kernel operates entirely in SRAM, with 32K words of program space in SRAM all toitself!

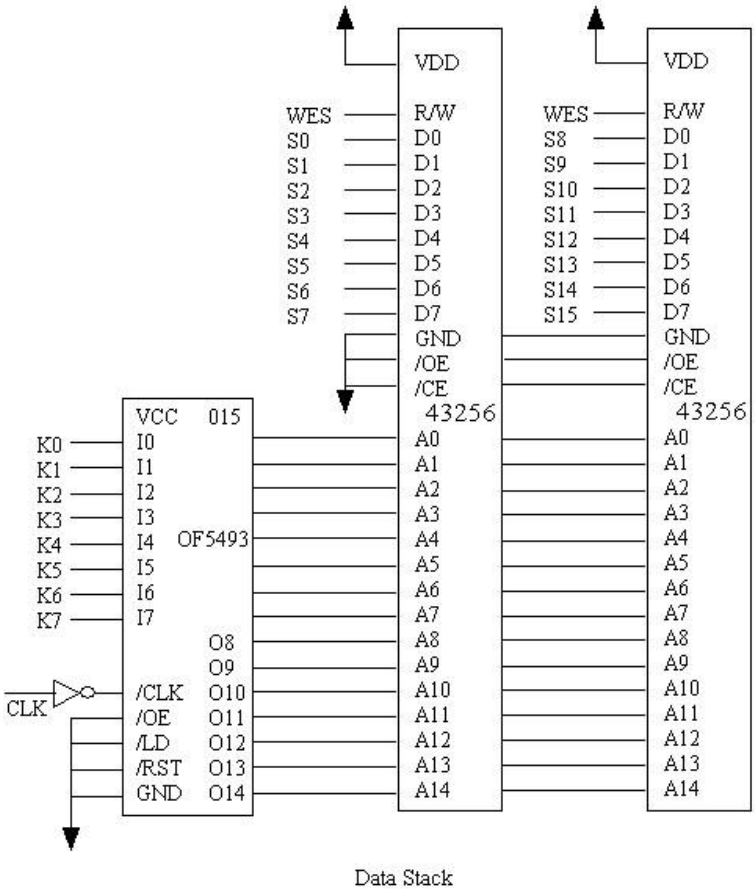


Figure 4.8. Expansion of Data Stack for NC 4000

A schematic diagram of this decoding logic is shown in Figure 4.9. The sequence of events during power-up or reset is as follows. Upon reset, the flip-flop 74HC74 is preset. /Q output is low which activates the PROM's, and NC4000 starts executing the boot-up code starting at 1000H. Part of the boot-up code copies the content of PROM's into SRAM at equivalent locations. OE or SRAM is driven high by the Q output of the flip-flop, so that the PROM's are read while SRAM's are written. After the kernel is copied, reading 8000H triggers the clock input of the flip-flop, lowering Q and raising /Q. Thus the PROM's are deactivated and the kernel in SRAM takes over the system. Further reading of 8000H does not affect the state of the flip-flop.

The code sequence performing this switch over is:

HEX 1000 7FF FOR 0@+ 1 !+ NEXT DROP 8000 @ DROP

This technique shows that a little bit of ingenuity can substitute for a lot of hardware.

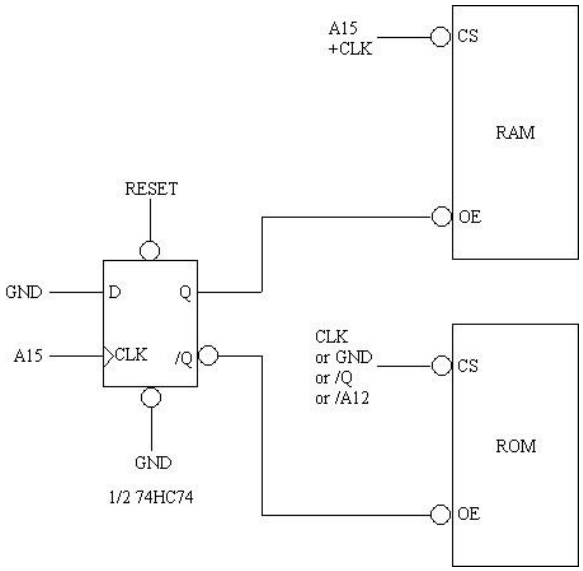


Figure 4.9. Decoding Memory with a 74HC74.

Chapter 5. The cmForth Operating System

This version of Forth, cmForth, was developed byChuck Moore, the chief architect of NC4000 Forth engine and the inventor of the Forth language. This version of Forth is installed in the EPROM's on ForthKit and Delta Board single board computer as its operating system. Chuck Moore and Novix kindly donated this remarkable software package to public domain to encourage people to explore the capability of NC4000.

This chapter serves as a documentation for cmForth. I will try to go through the system in minuet detail, in order to help people who are not familiar with the Forth language and 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 commentary to the source listings. The complete source listing of cmForth is included in this book as Appendix A. In the first two editions of this book, the source code was the version released in February, 1986. Recently, Chuck released a revised version in December 1987. This newer version is reproduced in Appendix A with shadow screen. The discussions in this chapter are also updated according to this new version of cmForth.

Chuck Moore had gone through the source code and made many modifications to the system. Although all the important features in the original cmForth are retained, there are enough changes making the new version quite different from the old one. Do not expect that program written under the old version will run under the new version. If you have used the old version for some time, it is probably better to keep on using it. To upgrade to the newer version, you have to go through you existing code carefully for words which were different between the two versions.

The source listings of cmForth are your best source for examples of code and programming style when striving to use NC4000 most efficiently. You are encouraged to study these listings carefully in order to get the most benefit from the chip.

5.1. The Kernel

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

5.1.1. The Primitive Forth Words

Primitive stack operators are machine instructions of NC4000 chip. However, it is necessary to give them names so that they will be available to the text interpreter for execution and compilation. In a conventional computer system they should be equivalent to assembler mnemonics which compile NC4000 machine instructions into a definition. If they are used in the following form during compilation as high level instructions, the performance will be degraded because of the overhead in nesting and un-nesting.

```
: 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 constructed from single NC4000 instructions and they have to be defined as high level Forth instructions.

```
: ROT           ( n1 n2 n3 -- n2 n3 n1 )
                PUSH SWAP               Exchange n1 and n2.
                POP SWAP                Exchange n1 and n3.
                ;
: 0=            ( n -- f )
                IF 0 EXIT THEN          Return false if not 0.
                                        EXIT is cheaper and faster.
                -1                      -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-                   An NC4000 primitive instruction.
                0<                      Extend the sign.
                ;
: =. _          ( n1 n2 -- f )
                XOR                     Compare all 16 bits.
                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 - PUSH             high - low
                -                       n - low
                POP U<                  In range?
                ;
: ABS           ( n -- u )
                DUP 0<
                IF NEGATE EXIT THEN     Invert negative number.
                ;
: MAX           ( n1 n2 -- n1 ; n2 )
                OVER OVER -             n1 - n2 0< IF
                BEGIN SWAP DROP         n1 < n2, drop n1. Otherwise, jump to THEN in
                                        MIN and drop n2.
                ;
: MIN           ( n1 n2 -- n1 ; n2 )
                OVER OVER -             n1 - n2
                0<
                UNTIL                   n1 > n2, jump to BEGIN in MAX and drop n1.
                THEN DROP               Otherwise, drop n1.
                ;
```

The funny IF-BEGIN... UNTIL-THEN structure spanning over two definitions MAX and MIN lets two definitions executing two alternate pieces of code, SWAP DROP or DROP. Chuck can do tricks like this because cmForth does not have compiler security and protection. Not recommended for general programming practice.

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

5.1.2. Memory Accessing Words

```
: +!            ( n a -- )
                0 @+                    Fetch from a, while keeping a on the stack.
                PUSH                    Save a.
                +                       Add n to content of a.
                POP !                   Store the sum back into a.
                ;
: 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                    Get n to the top.
```

```
                0 [ \\ ] +          Add 0 to n, thus clear the carry. \\ breaks 0 and + into two
                                    instructions.
                2/                  Unsigned divide by 2.
                ;
: C!            ( b a -- )          Store a byte to address a. a is a byte address, which has
                                    to be converted to a cell address.
                2/MOD DUP PUSH      Save cell address.
                @                   Cell content.
                SWAP                Byte offset.
                IF -256 AND         Offset=1. Mask off lower byte.
                ELSE 255 AND        Offset=O. Mask off higher byte.
                  SWAP              Get the byte b.
                  6 TIMES 2*        Shift left by 8 bits.
                THEN
                XOR                 Combine two bytes.
                POP !               Put the cell back.
                ;
: C@            ( a -- b )
                2/MOD               Get the cell address.
                @                   Content 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.
                ;
```

NC4000 is a 16 bit machine and it addresses the memory 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.

```
: -ZERO         ( al -- a1+1 a2)    A compiler directive to reduce one loop cycle to compensate
                                    for the extra loop in FOR...NEXT structure.
                1+                  Add 1 to the address where the loop begins.
                \ BEGIN             Copy current address a2 to stack.
                130000 ,            Compile an unconditional jump instruction here. The target
                                    address will be the last address in the FOR...NEXT loop so
                                    that activity in the loop can be skipped once.
                ;
: MOVE          ( a1 a2 n -- )      Copy n cells from al to a2. al is the starting address of the
                                    source, and a2 is the starting address of destination.
                PUSH                Push the loop index on return stack in place of FOR.
                4 I!                Save a2' in MD register.
                BEGIN               Complete the FOR instruction with PUSH. Leave current
                                    address on the data stack.
                 -ZERO              Compile an unconditional jump or ELSE here, and move the
```

```
                              loop address after it.
    1 @+                      Fetch a cell to the data stack.
    4 I@!                     Exchange al and a2.
    1 !+                      Store a cell to the destination.
    4 I@!                     Exchange al and a2 again.
  THEN                        This is where -ZERO skipped to in the first loop.
  NEXT                        Loop back to -ZERO+1 or 1@+ if n is not decremented to
                              zero.
  DROP                        Clear al, the last address.
  ;
: FILL    ( a # n -- )        Fill a memory range with cell value n.
  4 I!                        Save the value n in MD register.
  FOR                         Begin the loop.
    -ZERO                     Skip the loop once.
    4 I@                      Retrieve the stored value n.
    SWAP 1 !+                 Store it to destination.
  THEN                        For skipping by -ZERO.
  NEXT                        Loop back to 4 I@.
  DROP                        Discard last address.
  ;
```

5.1.3. Multiply and Divide

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

```
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!
  4 I!                        Store u2 in MD register.
  14 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.
  4 I!                        Store u in MD register.
  D2*                         Left shift d by 1 bit so that it is always even.
  13 TIMES /'                 Repeat divide step /' 15 times.
  /''                         Last divide step.
  ;
: M/      ( d u -- q )        Double integer d is divided by unsigned integer u.
```

```
                OVER 0<          Is d negative?
                IF
                  DUP PUSH       Save u.
                  +              Add u to the higher half of d.
                  POP            Retrieve u.
                THEN
                M/MOD            Do the divide now.
                ;
:               ( n1 n2 -- n3 n4 )  Negate top two integers on the stack.
VNEGATE
                NEGATE           Negate top integer.
                SWAP NEGATE      Negate next integer.
                SWAP
                ;
: M*            ( n1 n2 -- d )   Mixed mode multiplication of two signed integers.
                DUP 0<           Is n2 negative?
                IF VNEGATE       If so, negate both integers.
                THEN
                0  SWAP          Insert 0 into the accumulator.
                4 I!             Copy n2 to MD register.
                13 TIMES *'      Repeat multiply step.
                *-               Last signed multiply step.
                ;
: /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.
                ;
: */            ( n1 n2 u -- r ) Ratio of n1 xn2/u.
                PUSH             Save u.
                M*               Signed multiply of n1 and n2.
                POP M/           Divide by u.
                ;
: *             ( n1 n2 -- r )   Signed multiply.
                0 SWAP U*+       Signed multiply.
                DROP             Discard remainder.
                ;
: /             ( n u -- q )     Divide by unsigned integer.
                PUSH             Save divisor u.
                DUP 0<           Sign extend integer n.
                POP M/           Divide.
```

;

5.2. System Variables

System variables contain vital information needed by the 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 in single machine instructions. These are the most frequently used system variables. Less frequently used variables are kept above location 35.

Chuck eliminated VARIABLE in the target compiler in the new version of cmForth. The system variables are now defined as constants, returning the variable addresses. The behavior of the system variables are still the same.

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

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 in each disk buffer.
NB	A constant of value 1. Number of disk buffers less 1.
CYLINDER	Memory 20, not initialized. Cylinder of disk drive.
TIB	Memory 21, initialized to 36. Terminal Input Buffer.
SPAN	Memory 22, not initialized. Count of characters received from the terminal device.
>IN	Memory 23, not initialized. Pointer to the input stream of characters. Used by WORD to parse strings.
BLK	Memory 24, initialized to 0. Contains the block number under interpretation.
dA	Memory 25, initialized to 0. Memory address offset to be subtracted from the current address so that word address compiled can be relocated to other part of memory.
?CODE	Memory 26, initialized to 0. Storing the address of the machine code most recently compiled. Used by the optimizing compiler to construction multi-function instructions.
CURSOR	Memory 27, initialized to 0. Pointer to the memory location where input characters are stored.
SO	Memory 28, initialized to 1FFH Serial output polarity. It is either 1FFH or 200H.
BASE	Memory 29, initialized to 10. Number base for numeric I/O conversion.
H	Memory 30, initialized to 64 cells above terminal input buffer TIB. 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 serial RS-232 terminal port.
Interrupt Vector	Memory 32 and 33, initialized to POP DROP, a noop interrupt service routine.
Thread Table	Memory 34 and 35, initialized to 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. Storing the hash code of the context vocabulary.

5.3. Terminal Input and Output

5.3.1. Primitive Input and Output Words

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. For 5 MHz clock, C/B is 521. The primitive to send an ASCII character to the terminal is EMIT and that to receive a character from terminal is KEY. From these primitives, line based I/O words TYPE and EXPECT are defined.

```
HEX
: EMIT    ( c -- )          Send a character to the terminal via X0.
          1E D I!           Mask X-port to allow X0 to be output and other bits be input. 2*
                            SO @ XOR
          9 FOR             Send out 8 data bits with one start bit and one stop bit.
            DUP C I!        Send out one bit.
            2/              Shift out next bit.
            C/B @ A-        Wait for one bit period.
            CYCLES
          NEXT              Continue for the entire bit pattern.
          DROP              Discard the rest of the character.
          ;
: RX      ( -- n )          Get one bit from X4 pin.
          C I@              Read the X-port.
          10 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.
            10 XOR          Exit only when a start bit (low)
          UNTIL             is detected.
          C/B @             417 or 521 cycles per bit.
          DUP 2/ +          Wait 1.5 bit to the center of the first data bit.
          7 FOR             Read 8 bits.
            10 - 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.
        ;
```

5.3.2. 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. al is the starting cell
                            address and a2 is the address of the cell following the
                            string.
            2*              Change al to a byte address.
            DUP C@          Get the count byte.
            1 - FOR         Scan the string.
              1 +           Next character address.
              DUP C@        Get the character.
              EMIT          Send it out.
            NEXT
            2 + 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
                  CURSOR @  Store it in the assigned memory.
            1 !+
                  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.
                  POP       Get the current index.
                  -         Number of character received so far.
                  SPAN !    Store it in the character count variable SPAN.
                  EXIT      CR end of line exit.
                THEN        .
```

```
    ELSE                Yes. It is backspace.
      DROP              Discard the backspace character.
      DUP I XOR
      [ OVER ] UNTIL
      CURSOR @ 1 -      Get the cursor address again.
      CURSOR !          and decrement it.
      POP 2 + PUSH      Add 2 two loop index, and back up the index pointer
                        by 2.
      8 EMIT            Echo the backspace code.
    THEN
  NEXT
  1+ SPAN !             Increment character count.
  ;

  5.3.3.                Other Terminal I/O Words

: CR    ( -- )
  D EMIT                Carriage return.
  A EMIT                Line feed.
  ;
: SPACE ( n -- )
  20 EMIT
  ;
: SPACES ( n -- )
  0 MAX                 Protect security against negative count number.
  FOR                   Loop n+1 times.
    -ZERO               Skip the first loop.
    SPACE               Send them out.
  THEN                  -ZERO skips to here.
  NEXT                  Loop back to SPACE.
  ;
: HERE  ( -- n )
  H @                   Top of dictionary or the WORD buffer.
  ;
```

5.4. Number Conversion

5.4.1. Convert Digits to Binary Number

Strings of digits or numbers are one of the two basic syntactic elements in the Forth language. The numbers you typed in 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.
                 30 -                Take off the offset of ASCII 0.
                 DUP 9 >             Is c greater than 9?
                 IF                  Yes.
                   7 -               If so, subtract 7 to take care of the gap between 9 and A.
                   DUP A <           If the number is less than 10,
                   OR                make it a -1.
                 THEN                We have a number or a -1 at this point.
                 DUP BASE @ U<       Is the number less than the base?
                 IF EXIT THEN        If so, the conversion is successful.
                 Return with the
                 value.
                 2DROP               Otherwise, conversion error. Clear the stack.
                 ABORT" ?"           Abort with a message ?.
                 ; RECOVER           ; is need to terminate the compiler. It will not be executed
                                     because ABORT" exits to the interpreter. RECOVER
                                     reclaims 1 cell of memory.
: C@+            ( -- n )            Increment the byte address in the SR register. Use the
                                     address to fetch a byte from the main memory.
                 6 I@ 1 +            Get the address from SR and increment it.
                 DUP 6 I!            Return the new address to SR.
                 C@                  Fetch one byte from memory.
                 ;
: 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.
                 OE 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 number string at 'a' to a 16 bit signed integer.
                 BASE @ 4 I!         Store base value in MD register.
                 0                   Initial value of the number.
                 SWAP 2*             Convert 'a' to byte address.
                 DUP 1 + C@          Get the first character of the string.
                 20 = PUSH           Test for '-' and save the flag.
                 DUP 1 - 6 I!        Save the address of the first valid digit in the SR register.
                 C@                  Get the length of this string.
                 I +                 Correct it if the first character is a '-' sign.
                 1 - FOR             Scan the number string.
                   C@+               Fetch the next character.
                   10*+              Convert this character and accumulate the digit into the
                                     running sum.
                 NEXT                Repeat until all digits are converted. The accumulated
```

```
                        sum is left on the stack.
POP                     Retrieve the sign flag.
IF NEGATE               Negate the number if it has a leading - sign.
THEN
;
```

5.4.2. Convert Binary Number to ASCII String

This conversion process is different from what we know well in other Forth systems. The converted digits are piled up on the stack instead of stored in an output buffer. Using the stack to store the converted string is more efficient than using a buffer. In Forth, numbers are converted to ASCII strings only to be sent to the terminal or to a printer. The converted string is never needed internally. There is no reason to save the string in a permanently allocated output buffer.

```
: 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 PUSH                 Save n.
          SWAP                      Tuck c under #.
          1 +                       Increment count #.
          POP                       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 ASCII A.
          48 +                      Add offset of ASCII 0 to get the
          ;                         proper ASCII code of n.
: <#      ( n--#n )                 Prepare a number to start the conversion process.
          -1                        Initial value of the string length.
          SWAP                      Tug the count # under n.

: #       ( .. # n -- .. #' n' )    Convert one digit from n and add the converted digit to the
                                    output string.
          BASE 0 /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                     Repeat until n is reduced to 0.
          ;
: #>      ( . . # n -- )            Output the converted string to the terminal.
```

```
            DROP            n is usually 0 and not needed any more.
            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< NOP          Is n negative?
            IF 45 HOLD THEN If so, append - sign.
            ;
: (.)       ( n -- .. # )   Convert the number n to a ASCII string on stack.
            DUP PUSH        Save a copy of n for sign.
            ABS             .
            <# #S           Convert the absolute value to string.
            POP 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.
            ;
. U.R       ( u n -- )      Display an unsigned integer u in a field of n columns, right
                            justified. Formatted output.
            PUSH            Save the column number n.
            <# #S           Convert u to a string.
            OVER            Copy character count to top.
            POP 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.
            ;
```

5.4.3. 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 that at a. a+8 is returned on
                            the stack so another DUMP can be issued.
            CR              New line.
            DUP 5 U.R       Display first the address a.
            SPACE
```

```
7 FOR           Run down 8 cells.
  1 @+          Fetch one cell and increment a.
  SWAP 7 U.R    Display the content.
NEXT            .
SPACE           Add one space at the end of line.
;
```

5.4.4. Message Output

In an interactive programming environment, it is important that the system will send timely messages to the terminal to indicate to youits 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. I tried to collect as much information as possible here, toexplain the construction of these message output words. Some of the words used in these definitions will beelaborated later.

```
: COMPILE       ( -- )          Compile the address following this word to the top of the
                                dictionary.
                POP             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.
                PUSH            Put the address of the second word after COMPILE back
                                on the return stack.
                ,A              Compile the address of the next word into the dictionary.
                ;
: abort"        ( -- n 0 )      Run time routine for ABORT". Print the following
                                message and re-initialize the system.
                H @ TYPE        Display the name of word currently been executed.
                SPACE
                POP 7FFF AND    Address of the compiled text.
                TYPE            Display the message text.
                2DROP           Clear garbage left by TYPE.
                BLK @ ?DUP      Leave the block number on the stack for debugging aid.
                DROP
                0               A dummy 0 is compiled here. After QUIT is defined, this
                                word will be patched with the address of QUIT, which
                                returns control to the text interpreter.
                ;
: ABORT"        ( -- )          Abort the word currently been executed and return to the
                                text interpreter after displaying the following message.
                COMPILE abort"  Compile the runtime abort routine.
                22 STRING       Compile the following message up to " as a string literal.
                ;
: dot"          ( -- )          Run time routine of ." , which displays the message
```

```
                        immediately following until " .
        POP 7FFF AND    Address of the compiled message text.
        TYPE            Display the message and leave the address after the
                        message.
        PUSH            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" .
        22 STRING       Compile the following text up to " as a string literal.
        ;
```

5.5. Serial Disk

This implementation assumes 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. It requires a host computer at the other end of the RS-232 line and acts as the terminal and disk server to the cmForth system. Whenever a disk block is requested, the data will be transferred into cmForth through the serial link. When an updated block is flushed back to the disk, the data is also sent through the serial link to the host computer.

5.5.1. Disk Buffer Manager

Two disk buffers are maintained in cmForth. Each buffer is 1024 cells long. The first buffer starts at memory address 800H and the second buffer is at COOH below the ROM memory. Two cells at BUFFERS contain the block number associated with the data stored in the two buffers. The pointer PREV, points to the disk buffer which is referenced most recently. OLDEST points 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 in 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 surely become the PREV buffer and no disk access is necessary. If the requested block is not in the buffer, the manager now assigns the PREV buffer for the new block and this buffer contains data that is least recently referenced and is appropriate to be flushed to the disk or discarded.

This technique is often referred to as Ping-Pong buffers as 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.
                30 +            Two buffers are at memory locations
```

```
                8 TIMES 2*          30K and 31K.
                ;
: 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 0       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.
                POP 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.
                POP DROP            Discard the return address on top of the return
                                    stack, thus exit 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 in the least
                                    recently accessed buffer. If the block is not
                                    updated, skip the rest of 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 POP DROP DROP    skip the rest of the words following UPDATED
                THEN                by thrashing the return address of top of return
                                    stack. It is a very fast and dangerous 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 save the address of
                                    PREV.
                SWAP 32768 OR       Set bit 15, the update bit.
                SWAP !              Store it back to BUFFERS.
                ;
: 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 been most recently
                                    referenced.
                SWAP                Get n .
                PREV @ BUFFERS !    Store n into the PREV entry in the BUFFERS
                                    array.
                ;
```

5.5.2. Disk Read and Write

The serial disk is implemented using a very simple protocol. A disk read/write request is initiated by sending an ASCII NUL to the host at the other end of the RS-232 line, followed bytwo 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 disk read/write 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.
                    SWAP 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           Get one byte.
                    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-          ( -- )          Erase all the buffer pointers to make the disk
BUFFERS                           manager think the buffers are empty.
                  PREV            Address of the PREV variable.
                  [ NB 3+] LITERAL  The array including PREV, OLDEST, and
                                  BUFFERS.
                  0 FILL          Erase all these pointers to fool the disk manager.
```

```
        FLUSH                   Force the flushing of buffers.
        ;
```

5.6. Text Interpreter

The text interpreter is the operating system of Forth and it is the user interface which allows you 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 failed to convert the word into a number, the word is outside of the computer's vocabulary and it will stop executing the command line. It will then come back and ask you to type another command line and the process continues on forever.

The major functions required by the text interpreter are receiving command lines, parsing words, dictionary searching, command executing, 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 the text interpreter, and hence an operating system.

5.6.1. Parsing of Words

```
: LETTER    ( al a2 n -- a3 a4 )    Copy n characters from a2 to al. 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.
              6 I@ XOR              Is the character the same as the one stored in SR, the
                                    delimiter?
            WHILE                   Not equal.
              1@+ PUSH              Fetch character and increment a2. Save a2 on return stack.
                                    OVER C!
              1 +                   Increment al.
              POP                   Get a2 back.
            NEXT
            EXIT THEN               Exit if the string is completely copied.
            >IN @                   Now, process the character pointer.
            POP -                   Get the loop index off the return stack.
            >IN !                   Move the interpreter pointer >IN back that many characters.
            ;
. -LETTER   ( al a2 # -- a2 a4 )    Scan characters stored in buffer a2. Ignore leading
                                    delimiters by comparing with SR. Then move the string
                                    into buffer al. Again, al 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 6 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.
                  POP               Index of the do-loop when branched out at WHILE.
                  LETTER            Scan the rest of the string and copy it into al 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.
                PUSH                Save n, the delimiter.
                H @ DUP 2* DUP      Byte address of the destination string buffer. Leave one byte
                1 +                 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, you 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.
                  TIB @ +           Address of the character in buffer to be interpreted.
                  SPAN @            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.
                POP 6 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.
                SWAP C!             Store the count at the beginning of the parsed word as a
                                    packed count string.
                ;
```

5.6.2. Dictionary Search

To understand the words involved in dictionary searching, you have to know the structure of individual words in cmForth as well as how these words are arranged in memory to form a dictionary. A Forth dictionary is a linked list of words. A dictionary may be partitioned into many

vocabularies in which related words are grouped together.

Most Forth systems use indirect threaded code technique to construct high level definitions. A high level word is made to be equivalent to a list of addresses which point to memory locations containing addresses of executable code. The code field in a definition thus points to a piece of executable code, the inner interpreter, which executes this definition. Data or address list are stored in the parameter field following immediately after the code field.

cmForth uses directly threaded code technique, which eliminates the code field. The parameter field contains executable code and data if necessary. Subroutine calls are mixed with the machine code. The inner interpreters NEXT, NEST (DOCOL), and UNNEST (;S) are all native NC4000 machine instructions. A word defined in cmForth 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 long name is truncated and name comparison must stop at that cell.

The dictionary in cmForth is divided into two vocabularies: Forth, containing regular, executable Forth words; and COMPILER, containing compiler directives and assembler instructions which are used to compile new Forth words. The link field of a word points to the link field of the previous word in the same vocabulary. The first word in a vocabulary has a 0 in its link field, indicating the end of the linked list. The link field address of the last word defined in a vocabulary is stored in a vocabulary link table, immediately below the variable CONTEXT. The number stored in CONTEXT is used to select an entry in the vocabulary link table as the context vocabulary, which will be searched for words by the text interpreter. If CONTEXT contains 1, the Forth vocabulary will be searched. If CONTEXT contains 2, the COMPILER vocabulary will be searched.

HEX

: SAME	(al a2 -- al a2 f; a3 t)	With a string address al and the link field address a2 of a word in dictionary, compare the string with the word name. If the name matches the string, return the parameter field address a3 of the word found with a true flag. Otherwise, return the string address al, link field address a2 and a false flag. The character count of the string is stored in the SR register.
	OVER 4 I!	Save a copy of al in MD register.
	DUP 1 +	Copy a2 and increment it to get the name field address.
	6 I@ FOR	Scan both strings.
	1 @+	Fetch one cell from the name field.

```
          SWAP          Get its content to top of stack.
          4 I@          Address of a cell in the string.
          1 @+          Fetch one cell there.
          4 I!          Replace the string address.
          - 2*          Compare contents of the two cells, ignoring bit 15.
          IF            Name does not match, prepare to exit.
            POP DROP    Discard the loop index.
            0 AND       Clear the address and make it a false flag.
            EXIT        All done. Exit.
          THEN
        NEXT            Repeat comparing the strings.
        SWAP 1 + @      Examine the first cell in the name field.
        0< IF @ THEN    If bit 15 in the first cell of the name is set, this word used indirect
                        reference with separated head. Fetch the true parameter field
                        address. Otherwise, the parameter field address is on the data stack.
        SWAP            Get al and use it as a true flag.
        ;
: HASH  '               ( n -- a )
        CONTEXT         Subtract n from CONTEXT.
        SWAP-
        ;
        : COUNT         ( a -- n )
        7 TIMES 2/      Shift the upper byte down.
        15 AND          Allow only 15 cells.
        ;
: -FIND ( a1 n-- a1 t i a2  With word address al, link address a2 and vocabulary code n,
        f)              search the vocabulary for the word. If found, return the parameter
                        field addresses a2 of the word and a false flag. If not found, return
                        the word address al and a true flag.
        HASH            Find the head of the vocabulary.
        OVER @          Get the cell count of the string at al and save it in tne i'lu
        COUNT 6 I!      register.
        BEGIN           Start the dictionary search.
          @ DUP         Get the next link field address.
        WHILE           If link field address is not zero, continue the searching.
                        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        If found a match, put on the false flag and exit here.
        THEN            Reach the end of the vocabulary
        -1 XOR          without finding a matching name. Return with a true flag.
        ;
```

5.6.3. The Text Interpreter

```
: -'          ( n-- a1 t, a2 f)     Use vocabulary code n to search that vocabulary for the
```

```
                                  next word. Parse out a string from the input buffer and
                                  search the dictionary for a word with matching name.
                                  Return the parameter field address a2 and a false flag if a
                                  word is found. Return the word buffer address al and a
                                  true flag if the word is not found.
        32 WORD                   Parse out the next string in the input buffer, using ASCII
                                  BL as the delimiter.
        SWAP -FIND                Search through the context vocabulary for the word
                                  parsed out.
        ;
: '     ( -- a )                  Search the dictionary for the next word in the input buffer.
                                  Abort it not found. If it's found, return the parameter field
                                  address of the found word.
        CONTEXT @                 Get the vocabulary code of the context vocabulary.
        -'                        Searches that vocabulary.
        IF                        If the word is not in the dictionary,
          DROP                    Discard the address generated by -'.
          ABORT" ?"               Abort with a message.
        THEN
        ;
: EXECUTE  ( a -- )               Make a subroutine call to the code address a.
        PUSH                      Push the address on the return stack.
        ;                         When ; bit is detected by NC4000, it makes a subroutine
                                  jump to the address on top of the return stack, exactly what
                                  we want in EXECUTE.

: CYCLES   ( n -- )               Run n empty TIMES cycles. Just waste some time to wait
                                  for some other events.
        TIMES                     Waste n+4 cycles doing nothing.
        ;
: OCTAL    ( -- )
        8 BASE !                  Set base to 8 for octal input/output.
        ;
: DECIMAL  ( -- )
        10 BASE !                 Set base to 10 for decimal I/0.
        ;
: HEX      ( -- )
        16 BASE !                 Set base to 16 for hexadecimal 1/0 conversion.
        ;
: LOAD     ( n -- )               Execute text in block n.
        >IN 2@                    Get current >IN pointer and the currently used block
                                  number.
        PUSH PUSH                 Save then on return stack.
        0 INTERPRET               Process command text in the buffer which contains block
                                  n.
        10 BASE !                 Restore to decimal base always.
```

```
                POP POP >IN 2!      Restore >IN and BLK.
                ;
:               ( n1 n2 -- )        Use n2 as the interpreter pointer >IN and n1 as the BLK to
INTERPRET                           select input text buffer. Interpret the command text in the
                                    buffer.
                >IN 2!              Store n2 in >IN and n1 in BLK.
                BEGIN               Start the interpreter loop.
                  1 -'              Search the Forth vocabulary for the next word parsed out
                                    of the input buffer.
                  IF NUMBER         If it is not a defined word, convert it to an integer.
                  ELSE
                    EXECUTE         Execute the found word definition.
                  THEN
                AGAIN               Continue till the end of the buffer.
                ; RECOVER           The last word will never be executed. Recover the
                                    memory space.
: QUIT          ( -- )              The text interpreter. It accepts one line of commands from
                                    the terminal and executes the commands in sequence. If
                                    all the commands are executed without error, the message
                                    'ok' is displayed and it waits for next line of commands.
                BEGIN               It is an infinite loop.
                  CR                New line.
                  TIB 0 64 EXPECT   Accept one line (64 characters) from terminal and store it
                                    in the terminal input buffer at MSG.
                  0 0 INTERPRET     Execute the commands.
                  ." ok"            Print ok message.
                AGAIN               Continue forever.
                ; RECOVER           Save one cell.
```

5.6.4. Power Up-and Reset

There is a RESET pin on NC4000 chip. Whenever this pin is grounded and then released to 5 volts, the chip enters a reset cycle which starts by executing the instruction at memory location 1000H. A hidden word BOOT compiles the boot up routine staring here. Forthkit 4 uses a set of shadow EPROM's co-resident with SRAM's from 0 to 1FFFH. After power-up, BOOT in EPROM's is executed. It copies the cmForth object code from EPROM's to SRAM's and then turn the EPROM's off. After that, everything is run in the SRAM's. BOOT calls 'reset', which initializes everything and passes control over to the cmForth text interpreter QUIT.

We shall follow this train of events and explain the function of BOOT and 'reset' first. The low level words involved in the initialization will be presented afterwards.

```
{ : BOOT }      ( -- )          This headless word must be placed starting at memory
                                location 1000H so that cmForth can be booted up. It copies
                                itself from ROM memory chips to the SRAM chips in the
                                Forthkit system, turns ROM off, and executes the reset
```

```
                    routine Hereafter, only SRAM's are active.
16                  Address of the first system variable which needs to be
                    initialized--SPAN.
FFF FOR             Copy 4K words from ROM to SRAM.
  0 @+              Fetch one cell.
  1 !+              Store one cell and increment the address.
NEXT
-1 @                Pulse the 74HC74 flip-flop to turn off ROM and turn on
                    SRAM.
( reset) ;          This cell will be patched with the address of the 'reset'
                    routine which
```

Following BOOT is a table containing initial values of system variables which need to be initialized. Most of them default to zero.

```
0 , 0 , 0 , 0 , 0 , 0,          Filler for the memory locations 1009H to 100FH.
0.,                             PREV
0 ,                             OLDEST
0 , 0 ,                         BUFFERS
0 ,                             CYLINDER
0 ,                             TIB
77C0 ,                          SPAN
0 ,                             >IN
0 ,                             BLK
0 ,                             dA
0 ,                             ?CODE
0 ,                             CURSOR
1FF  ,                          SO
A ,                             BASE
0 ,                             H
DECIMAL 512 ,                   C/B

 { : interrupt }                Noop interrupt service
                  POP DROP      Discard interrupt return
                  ;             address.

                  0 , 0,        Vocabulary link table
                  1 ,           CONTEXT

 : reset          ( -- )        Send the Forth system into the QUIT loop to process input
                                commands from a terminal.
                  RESET         Trim the dictionary to the boot up state.
                  0 DUP 9 I!    Unmask B-port.
                  DUP A I!      B-port as 16-bit input port.
                  DUP B I!      B-port as latched output, no tristate.
                  DUP 8 I!      Clear B-port comparison latches.
```

```
                -1 A I!         Set B-port as output port.
                DUP D I!        Unmask X-port.
                DUP E I!        X-port set to be input port.
                F I!            X-port as latched output, no tristate.
                1A C I!         Set X1, X3 and X4 high.
                TIB 20 XOR      Compare 2 cells at TIB, if they are not equal, do a cold
                                boot. If they are equal, do a warm boot.
                IF              Not equal. Cold boot.
                  EMPTY-
                BUFFERS         Clear the disk buffers.
                  SPAN @
                TIB !           Copy 77COH in SPAN to TIB, which defines the terminal
                                input buffer.
                THEN            Skip above if doing a warm boot to preserve the disk
                                buffers.
                RS232           Observe the polarity of the serial input line and set the
                                serial output polarity SO accordingly.
                F E I!          XO serial output and XO serial input.
                BPS             Wait for an ASCII B from the serial input line to set the
                                baud rate.
                ." hi"          Send the sign-on message to the terminal. Ready to operate.
                                QUIT
                ;
: RS232         ( -- )          Determine the polarity of the serial input line and use it the
                                set the input and output polarity of the X4 and XO lines. It
                                allows Forthkit 4 to adapt automatically to a large number
                                of ASCII terminals.
                RX              Read X4 line.
                IF EXIT         If X4 is high, everything's been set correctly already.
                THEN
                200 SO !        Otherwise, reverse the output polarity.
                OB C I!         Also reverse the input polarity.
                ;
: BPS           ( -- )          Wait on the serial input line X4 for a character 'B' to
                                determine the baud rate of the terminal. Adjust the variable
                                C/B so that cmForth can talk to terminals using any
                                reasonable baud rate.
                4               Starting value for C/B.
                BEGIN           Wait until X4 is high.
                RX 10 XOR
                UNTIL
                BEGIN           Now, count cycles till X4 goes low.
                  5 +           Increment count by 5 cycles per loop.
                RX UNTIL        Stop counting if X4 drops low.
                2/ C/B !        Two high bits at the beginning of 'B'. A start bit and the
                                most significant bit are glued together.
                ;
```

```
: FORGET        ( -- )          Remove all definitions added above the dictionary in
                                cmForth to start over.
                POP 7FFFH       Get the address on the return stack, which points to the
                AND             memory immediately after FORGET in a definition, see
                                RESET for example.
                DUP 2 +         RESET is the last definition in cmForth. 2 cells after
                                FORGET in RESET is the free space for the dictionary to
                                grow.
                H !             Set the dictionary pointer so that new definitions can
                                be compiled above RESET.
                2@              The cells after FORGET in RESET store the initial
                                vocabulary link table. Fetch link field addresses of last
                                words in Forth and COMPILER.
                CONTEXT 2 -     Initialize the vocabulary link table below variable
                2!              CONTEXT.
                1 CONTEXT !     Make Forth the context vocabulary.
                ;               Return to the word after RESET in 'reset', because the return
                                address from FORGET was popped off the return stack and
                                the next return address points to the phrase 0 DUP 9 I! in
                                'reset'. Chuck Moore is the only person privileged to do this
                                kind of aerobatics. Not recommended for the rest of us.
: RESET         ( -- )          The last definition in cmForth. It contains the FORGET
                                mechanism to restore the dictionary to the boot up state.
                FORGET          Chop the dictionary down to its original size by re-initialize
                                the vocabulary link table and CONTEXT.
                0               2 cells here contains the link field addresses of the last
                                words defined in Forth and COMPILER vocabularies.
                ;
```

5.7. Compiler

The compiler loop is very similar to the text interpreter loop, in that it scans the input buffer to parse out words whose addresses are then added to the top of the dictionary, which is the parameter field of a new word definition. When a word cannot be located in the dictionary, the compiler will try to convert it to a number and compile the number into the dictionary as a literal. A special class of words, the compiler directives, are not compiled but executed inside the compiler loop. These compiler directives take care of many conditions which have to be dealt with immediately, like construction of branching and looping structures, compiling various types of literals, etc. All compiler directives are collected in a special vocabulary COMPILER and are not available outside of the compiler loop.

5.7.1. Compiler Loop

We shall begin by showing the high level definitions of the compiler] and its companion [. The low level words used by them and other words supporting compilation of new words are discussed later in this section.

```
OCTAL                          Machine code is best shown in octal.
: [          ( -- )            The compiler loop.
             BEGIN             Start an infinite loop.
              2 -'             Parse next word out of the input buffer and search the
                               COMPILER vocabulary first. If the word is in the
                               COMPILER vocabulary, it is executed. Otherwise, do
                               the following.
              IF 1 -FIND       If the word is not in the COMPILER vocabulary, then
                               search the Forth vocabulary.
                IF             If the word is not in either vocabulary, then
                  NUMBER       convert it to an integer
                  \ LITERAL    and compile it as a literal.
                ELSE           The word is found in the Forth vocabulary. Find the best
                               way to compile it.
                  DUP @        Fetch the first cell in the parameter field of the found
                               word to determine if it can be compiled as a machine
                               instruction.
                  DUP          Make a copy because we have to do a few comparisons.
                  140040 AND   If it is an I/0 instruction with return bit set
             140040 =
                   OVER        but not a variable,
             170377 AND
             140342 XOR
             AND
                  SWAP 170040  or an ALU instruction with return bit set,
             AND 100040 =
             OR
                  IF           then this instruction can be assembled directly in-line.
                    @ 40 XOR   Turn off the return bit because it cannot be assumed to be
                               the last instruction.
                    ,C         Compiled as a machine code.
                  ELSE         It cannot be compiled as a single machine instruction.
                    ,A         Compile it as a subroutine call.
                  THEN
                THEN
              ELSE EXECUTE     The word was found in the COMPILER vocabulary. It
                               must be executed immediately.
              THEN
             AGAIN             Infinite loop.
             ; RECOVER         There is no end it it.
: [          ( -- )            Exit the compiler loop and return to the interpreter loop.
             POP DROP          The compiler loop is set up inside the definitions of ].
                               Executing [ inside the compiler loop discards the return
                               address on the return stack. When ; is executed, control is
                               given back to the word calling [, which is EXECUTE in
```

```
                                  the text interpreter loop. The text interpreter will then
                                  start executing the words after [ until ] starts the
                                  compilation again.
                ;                 [ must be executed, not compiled. It must be compiled
                                  into the COMPILER vocabulary.
: ,             ( n -- )          This word 'comma' is the compiler in the most primitive
                                  form. It adds a 16 bit number to the top of the dictionary
                                  and increments the dictionary pointer. All other compiler
                                  words are derived from 'comma'.
                H @               Get the dictionary pointer.
                !                 Store n to top of dictionary.
                1 ALLOT           Move the dictionary pointer passing the compiled pattern.
: ,C            ( n -- )          Compile n as a machine code.
                H @ ?CODE !       Store the address of the code
                ,                 in variable ?CODE for optimization.
                ,                 Compile n.
                ;
: ,A            ( a -- )          Compile an address.
                dA @ -            Subtract the offset address store in dA, to facilitate
                                  building a target image in a virtual memory array.
                ,C                Compile the virtual address as a subroutine call.
                ;
: LITERAL       ( n -- )          Compile a number as a literal. The number will be
                                  pushed on the stack at run time.
                DUP -40 AND       Is n greater than 31?
                IF                Yes. Compile a 16 bit literal.
                  147500 ,C       Compile a literal fetch instruction first.
                  ,               Then compile the 16 bit number.
                  EXIT            EXIT is faster than ELSE.
                THEN
                157500 XOR ,C     Insert n into the short literal fetch instruction and
                                  compile it.
                ;
: \\            ( -- )            Break the process of compiler optimization. Complete the
                                  last word compiled and start a new word.
                0 ?CODE !         Clear the variable ?CODE. The next word will be
                                  compiled fresh.
                ;
                : ALLOT           ( n -- )
                H +!              Allocate n cells in the dictionary by moving the
                                  dictionary pointer H.
                \\                Start compiling a new word.
                ;
: PREVIOUS      ( -- a n )        Return the name field address and the count of the name
                                  field of the most recently defined
                word.
```

```
                CONTEXT @       Pointer to the link field.
                HASH
                @               Link field address of last word.
                1 +             Name field address.
                0 @+ SWAP       Fetch first cell while retain the name field address.
                ;
: COUNT         ( n1 -- n2 )    Extract the length of name from the first cell in the name
                                field.
                7 TIMES 2/      Right shift n by 8 bits.
                15 AND          Half of the character count.
                ;
: USE           ( a -- )        Replace the first cell in the parameter field of the last
                                word by the address ¯i, which will call an inner
                                interpreter.
                PREVIOUS        Get the name field address and the first name cell of the
                                last word.
                COUNT + 1 +     parameter field address.
                !               Replace the code with a.
                ;

HEX                             Best for bit counting.

: DOES          ( -- )          Used to define an inner interpreter for a class of words.
                                The syntax is:
                                : <name> CREATE <compiler>
                                DOES <interpreter> ;
                POP             Get the address of the next word, which starts the inner
                                interpreter.
                7FFF AND        Strip the MSB bit, which is carry.
                USE             Compile that address into the parameter field of the
                                newly defined word as its inner interpreter.
                ;
: SMUDGE        ( -- )          Set the smudge bit in the name
                of the last word.
                PREVIOUS        Return the name field address
                and the first cell in
                the name.
                2000 XOR        Set the smudge bit.
                SWAP !          Put it back.
                ;
: EXIT          ( -- )
                POP DROP        Pop the return stack and return to the caller.
                ;
: COMPILE       ( -- )          At run time, compile the next word in a definition.
                POP 7FFF AND    Get the address of the next word from the return stack.
                1 @+            Fetch the code in that cell. Increment the address also.
```

```
                PUSH            Push the incremented address back on the return stack to
                                skip the next instruction.
                ,A              Compile the next instruction on top of the dictionary.
                ;

COMPILER        OCTAL

: EXIT          ( -- )          This is the EXIT to be compiled inside a definition, not to
                                be executed interactively.
                100040 ,C       Compile the return instruction.
                ;               Perform the compiling function immediately.

HEX

:               ( -- )          Reset the smudge bit so that the word being compiled is
RECURSIVE                       made available to do recursive programming.
                PREVIOUS        Get the name field address and the first name cell.
                DFFF AND        Reset the smudge bit.
                SWAP !          Put it back.
                ;
: ;             ( -- )          Terminate a colon definition.
                \ RECURSIVE     Not to do recursion. Just clear the smudge bit in the
                                current word definition, making it available for searching.
                POP DROP        Pop return stack. Return to the caller at next ; .
                \ EXIT          Compile the 100040 return machine code.
                ;               Must be executed, not compiled.
```

5.7.2. Defining Words

In this cmForth system using NC4000 chip, code field loses its significance due to the fact that machine code can be mixed with high level subroutine calls. However, we can still place a subroutine call as the first cell in the code/parameter field and use it to execute a specialized interpreter for a class of words, very similar to the CREATE..DOES> structure in conventional Forth system.

```
: CREATE        ( -- )          Use the next word in the input buffer to create a
                                new header in the dictionary. Initialize the word to
                                act like a variable, returning the address of second
                                cell in the code/parameter field.
                H @ 0 ,         Save the link field address and compile a dummy
                                link field.
                40 WORD         Parse out the next word in the input buffer.
                CONTEXT @ HASH  Find the address of the head of thread of the linked
                                chain to which the new word will be linked.
                2DUP @          Fetch the link field address of the last word in this
                                chain.
```

```
                SWAP !              Store it in the link field of the new word.
                SWAP 0 COUNT 1 +    Find the cell length of the name field and allocate
                ALLOT               that many cells to the name field.
                !                   Attach the truncate bit to the last cell of the name
                                    field.
                147342 ,            Compile PC I@ ; , which pushes the integer in the
                                    next cell on the data stack.
                ;
: :             ( -- )              The definition of : in the target system.
                CREATE              Create a new header for this colon definition.
                -1 ALLOT            Colon words are the natural, default word type in
                                    cmForth No inner interpreter is necessary. Reclaim
                                    the cell used by CREATE.
                SMUDGE              Set the smudge bit to protect this definition.
                ]                   Call the compiler to compile the rest of the colon
                                    definition.
                ;
: CONSTANT      ( n -- )            Create a new integer constant.
                CREATE              Build a new header.
                -1 ALLOT            Reclaim the code field.
                \ LITERAL           Compile n as a literal, long or short.
                \ EXIT              The ; bit must be compiled as a separated cell.
                ;
: VARIABLE      ( -- )              Create a new variable.
                CREATE              That's all we have to do,
                0 ,                 except that the value of the variable must be
                                    initialized to zero.
                ;
: ARRAY         ( n -- )            Define a vector array, with n cells. This defining
                                    word is defined as a part of the target compiler.
                CONSTANT            Define the array as a constant, which also stores
                                    the dimension n .
                154462 USE          Compile the inner interpreter, which fetches the
                                    array address and add an offset to the array base
                                    address to return the correct
                ;
```

5.7.3. Control Structures

NC4000 has three branch instructions: unconditional branch, conditional branch and loop. A branch instruction takes a 12-bit argument to specify the address to be jumped to, within a 4K word pages. 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. cmForth supports the following:

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

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

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.

When n FOR ..NEXT is executed, the loop will be repeated n+1 times as the loop index is decremented from n to 0 before exiting the loop. If only n loops are desired and 0 FOR ... NEXT will skip the loop completely, Chuck gave use a new fix:

FOR … -ZERO ... THEN ... NEXT

What is between -ZERO and THEN will be repeated n times and that between THEN and NEXT will be repeated n+1 times.

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, which does not restore the return stack.

: \	(--)	Compile the next COMPILER word, which normally will be executed in a colon definition.
	2 -'	Parse out the next word and search it in the COMPILER vocabulary.
	IF DROP	If the word is not in the COMPILER vocabulary, abort
	ABORT" ?" THEN	immediately.
	,A	The word is found in the COMPILER vocabulary, compile it here. When the colon word containing it is executed, this word will then be executed in its turn.
	;	

. OR,	(a n --)	OR the address a into the instruction n and compile the branch instruction.
	\\	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)
	H @~	Push the current dictionary pointer on the data stack.
	\\	Initialize the optimizer.
	;	
: 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 conditional branch instruction on the stack.
	110000 ,	Compile a conditional branch instruction with an unresolved address field.
	;	
: ELSE	(a1 -- a2)	Resolve the conditional branch instruction at al. 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 al to top of the stack.
	\ 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 pass the
                                    address left by BEGIN.
                \ IF                Invoke IF to compile a conditional branch.
                SWAP                Exchange al 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.
                \ AGAIN             Compile an unconditional branch back to BEGIN, using
                                    address a2.
                \ THEN              Resolve the conditional branch instruction compiled by
                                    WHILE.
                ;
: FOR           ( -- a )            Start a definite loop.
                \ PUSH              Compile a PUSH instruction which saves the loop count
                                    in the I register.
                \ BEGIN             Leave address of the next instruction for NEXT to branch
                                    back.
                ;
: NEXT          ( a -- )            Compile a loop instruction and use the address a on the
                                    stack for the branch address.
                120000              Code of the loop instruction.
                OR,                 Resolve the backward jump address.
                ;
```

5.7.4. NC4000 Assembler

Assembler? Good grief!

Supposedly, NC4000 speaks high level Forth language and we shall all be free from the tyranny of assembler and live happily ever after. The truth is that you can program in Forth and NC4000 will run the program much faster than anything you had previously. However, if you really want the best out of this machine you still have to deal with it, by bits and pieces at the machine code level. If you know how to construct machine code which performs the task in the most efficient way, you can squeeze the most out of this machine.

NC4000 machine instruction assembly can be handled in two different ways: map 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 NC4000 instructions as possible in order to save both machine cycles and memory space. Here we shall be concerned with single function NC4000 instructions and show you how they can be defined and how they are used to allow us to program in regular Forth style. In the sections on the Optimizing Compiler, we will discuss how a program can be optimized by combining many functions into one NC4000 instruction.

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

Most of NC4000 machine instructions can be defined using uCODE. Here are all these words defined this way in cmForth:

```
100000 uCODE NOP        One cycle Noop.
140000 uCODE TWO        Two cycle Noop.
154600 uCODE O+c        Adjust for carry.
102404 uCODE MOD'       Conditional subtract MD.
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.
102412 uCODE *F         Fractional multiply step.
102416 uCODE /'         Divide step.
102414 uCODE /"         Last divide step.
102412 uCODE *F         Fraction multiply step.
102616 uCODE S'         Square-root step.
```

Machine code can also be defined as regular Forth words which compile specified code into the dictionary using C.

```
: R>        147321 C ;
: POP       47321 C ;
: PUSH      157201 C ;
: I         147301 C ;
: TIMES     157221 C ;
```

Instructions which use the least significant 5 bits for short literals, internal register numbers, and memory increments must compile proper values into this 5 bit field. The technique in defining them might be useful in other places.

OCTAL

```
. -SHORT        ( -- f )          Return a true flag if the current instruction under
                                  construction takes 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?
                ;
: 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 a 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.
                ;

                COMPILER          Assembler instructions have to be placed in COMPILER.

: @             ( -- )            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.
: !             ( -- )            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
```

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 prior to these instructions.

```
: 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
                ;
```

5.7.5. Compiler Vocabulary

To program in cmForth, you have to be aware of the difference between the compiler directives and regular Forth words, which can be compiled and interpreted. They appear to be the same, in a colon definition 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 after the word : is executed will the COMPILER vocabulary be made available to the compiler, which will take advantage of NC4000 and compile efficient machine code whenever possible. At the end of a definition or

when an error occurred, the COMPILER vocabulary will be turned off so that you will be protected from the abnormal behavior in many of the 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 one of the two threads to link a new definition and to search for an existing definition.
	;	
: COMPILER	(--)	Define the COMPILER vocabulary.
	2 CONTEXT !	The hash code of COMPILER vocabulary is 2. The compiler searches this vocabulary and execute words found here. If then searches Forth vocabulary and compile words found there.
	;	

5.8. 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 compiler directives during compilation. The compiler for NC4000 is much more complicated due to following reasons:

- The compiler absorbs the function of an assembler to assemble machine instructions besides compiling high level words or subroutine calls.
- More than one function may be performed by a single NC4000 instruction. The compiler must be able to recognize the sequence of actions 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 cmForth, Chuck Moore chose a very simple and quite effective approach to optimize the assembly of machine instructions. He simply looks at the last instruction just compiled and the current instruction. If there are unused bits in the last instruction which can accommodated the current instruction, the current instruction is then combined into the last instruction. If it is impossible to squeeze the current instruction into the last compiled one, a new instruction is compiled, which can be used to optimize the next instruction. The system variable ?CODE points to the last compiled instruction to facilitate this optimization process. A zero in ?CODE forces the compilation of a new instruction.

Chuck picked several strategic places 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.

The 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 compilers 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 compilers off for that instruction.

5.8.1. Smart ; Compiler

The subroutine call in NC4000 is a one cycle instruction and the subroutine return is a single bit embedded in many NC4000 machine instructions. Obviously, if you can recognize the conditions when the return bit can be inserted into the last instruction of a definition, you 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, return bit should not be
                              packed into it. Exclude memory instructions and return
                              stack instructions.
          IF                  Bit pattern does not match 140201,
            40 SWAP +!        pack the return bit into the instruction at address a.
          ELSE                Pattern matches with 140201.
            DROP              Discard the address a.
            100040 ,          Compile an explicit return instruction.
          THEN
          POP 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.
            \\                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  Is it an ALU instruction?
          100000 =
              IF PACK THEN    Yes. Pack the return bit.
               DUP 170300 AND Is it a register fetch instruction?
          140300 =
              IF PACK THEN    Yes. Pack the return bit.
               DUP 170000 AND Is it a short literal store instruction?
```

```
            150000 =
              IF
                DUP 170600 AND      15x6xx cannot be a valid instruction.
            150000 XOR
                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 explicitly return
                                    instruction. This is what computer scientists call a tail
                                    recursion.
              DUP HERE dA @ -       Compare the address in ?CODE with the current
            XOR                     dictionary pointer.
              170000 AND 0=         Are they in the same 4K word 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 to by ?CODE.
                EXIT                Terminate here immediately.
              THEN
              DROP                  Discard content of ?CODE.
             THEN
            THEN DROP               Discard ?CODE.
            100040 ,                Compile explicit return instruction. Not possible to
                                    optimize.
            ;                       Compiler directive.
: ;         ( -- )                  The optimizing ; compiler.
            \ RECURSIVE             Reset the smudge bit in the name field of the new
                                    definition, making it available for searching.
            POP DROP                Exit the compiler loop at the end of this word (;).
            \ EXIT                  EXIT was made immediate. Force its compilation.
            ;
```

5.8.2. Smart ALU Function Compiler

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

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

```
: BINARY    ( n1 n2 -- )            n2 is the code of an 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 parameter field.
DOES                Now define what the new compiler directive will do
                    during compilation.
POP 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     Is it of the SWAP/OVER type?
107100 =
 OVER 177700        Or a short literal?
AND 157500 = OR
 IF                 Yes. We can do something about 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      Test if carry must be included. IF 500 XOR
200 =
  ELSE
   DUP 70000        Make sure we have an ALU instruction at hand, then
AND 0=
   IF 20 XOR        flip the Stack Active SA bit.
THEN
   THEN
   ?CODE @ !        The instruction can take ALU code in the ALU field.
EXIT                Update the machine code.
  THEN
 THEN
THEN
DROP                Drop the ?CODE, which is zero
,C                  Compile n2 as another ALU instruction without
                    optimization.
DROP                Discard the compare mask.
;
```

Now, all 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 –
7100 107020 BINARY DROP

5.8.3. Shift Compiler

Shift instruction can be appended to all the ALU instructions. However, restrictions in NC4000 prototype chip have to be imposed so that shifts produce the desired results.

```
: SHIFT     ( n1 n2 -- )          Define smart shift compilers. n1 is the shift code and n2
                                  is a mask for comparison.
            CREATE                Make new header.
            ,                     Save n's, the shift code and a mask in the parameter field.
            DOES                  Actual compilation action.
            POP 77777 AND         Pointer to the stored shift code.
            2@                    Get the code and the mask.
            ?CODE @ ?DUP          Is the previous word a machine instruction?
            IF                    Yes. Do optimization.
              @ AND               Put on the mask.
              100000 =
            WHILE                 Is it an ALU instruction save with a shift operation?
              ?CODE @ +!          Yes. Pack in the shift code.
              EXIT                Done and out.
              THEN
            THEN
            DROP                  Cannot optimize. Discard the code address.
            100000 XOR ,C         Compile a simple shift machine instruction.
            ;
```

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

2 171003 SHIFT 2*
1 171003 SHIFT 2/
3 177003 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 evident in the source listing when 0< is invoked.

The double integer shift instructions 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/

5.8.4. 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

before the machine instruction just compiled.

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

5.9. The Target Compiler

The target compiler is a utility program in Forth which allows a Forth system to generate a new Forth system, to be run either on the same computer or on a different computer. In cmForth, the target compiler is the most important application written for a NC4000 computer, allowing cmForth to regenerate itself. This unique feature permits a user to modify cmForth, add new features, delete features not needed in his application, and produce an application or a system best suited for his purpose. cmForth can thus grow with you and your application. A user is not constrained by the prejudice and preference of its author, who does have his own ways in programming and shows no respect for conventional wisdom.

5.9.1. Utility Compiler

```
: 0<        ( -- )              Compile 0< with a NOP before doing logic decision. This
                                is necessary for prototype NC4000, which does not allow
                                enough time for the sign bit to propagate through the
```

		shifter.
	\ 0<	Forcing 0< to be compiled.
	\ NOP	Forcing NOP to be compiled. Allow sign bit to propagate 15 bits.
	;	
: END	(-_)	Very similar to ; at the end of a colon definition. Un-smudge the current word under construction and return to the caller of the current word in run time.
	\ RECURSIVE	Un-smudge the name of current word.
	POP DROP	Discard the top element on the return stack. Skip all words following the current word and return to the caller.
	;	
: REMEMBER;	(--)	A compiler directive which saves the current vocabulary link table so that the dictionary can be reduced to this point.
	CONTEXT 2-2@	Fetch the vocabulary links.
	, ,	Copy the current thread table into the parameter field.
	\ END	Compile EXIT and un-smudge the new word which will cut back the dictionary.
	;	

FORTH

: EMPTY	(--)	Create a new word named EMPTY. The overlay starts here. You can load an application package and do some useful work. After you are through with this application, execute EMPTY to remove the package from the dictionary and reclaim all the dictionary space for your next application.
	FORGET REMEMBER;	REMEMBER; stores the current vocabulary links in the two cells right after FORGET. When EMPTY is executed, FORGET will copy these two cells into the vocabulary link table below CONTEXT. All words defined after EMPTY will not be found by either the interpreter or the compiler.
: THRU	(n1 n2 --)	Load blocks from n1 to n2 inclusive.
	OVER -	
	FOR	Repeat that many times.
	DUP LOAD	Load one block.
	1+	Add one to n1 for the next block.
	NEXT	Repeat.
	DROP	Discard n1.
	;	
: -MOD	(n1 n2 -- n3)	Conditionally subtract n2 from n1. If the result is positive, return it as n3. Otherwise, return n1 unmodified.

```
4 I!            Copy n2 into MD register.
MOD'            Do the conditional subtraction.
;
```

5.9.2. Target Dictionary

Since this purpose of this target compiler is to regenerate cmForth, it is quite simple because most of the compiler functions are already implemented. What we need in addition are a set of words which will let us compile code in some unused part of memory, which will later be dumped into ROM's for the target computer. A special variable H' is defined to manage the target dictionary in the virtual memory space. Another variable dA is used to store an address offset, which is the displacement of the virtual memory address to the actual address in the target system.

VARIABLE H'	The dictionary pointer for the target system dictionary. It behaves very similarly to the system variable H, the dictionary pointer of the host system.
HEX 2000 ,	This cell following H' is reserved to store the dA variable of the target system.
2000 800 0 FILL	The target dictionary is compiled from 2000H to 27FFH. This space is first cleared to zero's. The code compiled here will be moved to ROM which will occupy memory 0 to 7FFH in the target system.
2000 H' !	H' is initialized to 2000H. Code of the target system will be compiled starting here.

Many words in cmForth are needed to construct the system and are of very little interest to the user. It is a waste of memory to keep their headers in the target dictionary. The following words allow us to compile only the body of a definition to the target dictionary. By keeping the header in the host dictionary, these words are still available to build other words in the target dictionary.

```
: {             ( -- )          Compile the header of the next word only to the host
                                dictionary.
                dA @            Get dA of the host system.
                HERE            and the dictionary pointer.
                H' 2@           Get dA and target dictionary pointer.
                H ! dA !        Store H' and dA'.
                H' 2!           Store H and dA. Exchanging H and dA of the host with those
                                of the target system allows cmForth to compile words either to
                                the host dictionary or to the target dictionary.
                ;
: }             ( -- )          Alias of {. It is defined for syntactic clarity. It is always used
                                when H' is pointing to the host dictionary, thus allows the
                                compiler to resume compiling the target dictionary.
                {               Alias of {,
                ;

                COMPILER        We need two copies of }, one for the text interpreter as above,
```

```
                                and another for the compiler which can be executed inside a
                                colon definition.

 : }            ( -- )          The compiler directive }.
                H' @            First get the target dictionary pointer,
                ,A              and compile it into the host dictionary so that this address can
                                be compiled to the target when it is invoked in a target
                                definition.
                \\              Break the optimization process before switching dictionary.
                PREVIOUS        Name field address of the word under construction and the first
                                cell in the name.
                8000 XOR        Set the MSB high and flag it as a hidden name of a target
                SWAP !          definition.
                {               Now, switch dictionary pointer so that subsequent words are
                                compiled into the target dictionary.
                ;

FORTH

 : forget       ( -- )          Hide the target definition which collides with the same
                                definition in the host dictionary which must be used during
                                target compilation.
                SMUDGE          Set the smudge bit in the name of the target definition.
                ;
 : RECOVER      ( -- )          Move the dictionary pointer H back by one cell. Save one cell
                                in the target dictionary.
                -1 ALLOT        Decrement H pointer.
                ;
```

5.9.3. Variables in Target System

In the earlier cmForth systems, Chuck Moore had to define many words to manage variables in the target system during target compilation. In this version, he simply replaces the variables by constants. The value returned by these constants are addresses pointing to memory where variables are stored. The variable array RAM, the variable pointer R', and the target version of VARIABLE are all eliminated. The newer cmForth is thus simpler and easier to understand.

5.9.4. Separate Target and Host Dictionary

Using { and }, we can compile the header of a target definition either in the target or the host dictionary. The resulting vocabulary link runs like a bowl of spaghetti, up and down between the host dictionary and the target dictionary. The link field in the target dictionary must be re-ordered so that the target dictionary can be searched when it is moved into a target system. Smudge heads in the target dictionary also must be un-smudged. The following words are used to re-link and clean up the target dictionary.

: SCAN	(a1 -- a2)	Following the linked dictionary chain starting at al until a link address outside of the host dictionary is found, which is returned as a2.
	@	Link field address of the next definition in the chain.
	BEGIN	Follow the link.
	DUP 1 2000	Is this address inside the host dictionary?
	WITHIN	
	WHILE @	Yes. Fetch the next link.
	REPEAT	Exit if the link address is either above 2000H in the target dictionary, or 0 which indicates the end of the vocabulary link.
	;	
: TRIM	(al a2 -- a2)	Relocate a target definition from the host dictionary to the target dictionary. Un-smudge the name of the target definition also.
	DUP PUSH	Save the link field address of the target definition a2.
	dA @ -	Compute the correct target address of a2.
	SWAP !	Store the correct target address into the link field of the target definition at al.
	POP DUP 1 +	Get the link address a2 back. Increment it to the name field of this target definition.
	DUP @ DFFF AND	Mask off the smudge bit in the name.
	OVER !	Store the un-smudged name back.
	DUP @ 200 / F AND	Get the cell count of the name field.
	+	The last cell in the name field.
	DUP 0 FF7F AND	Erase the most significant bit in the last byte of the name.
	SWAP !	This is the truncation bit.
	;	
: CLIP	(a --)	Scan a linked vocabulary for words in the target dictionary. Re-link them into a separate target vocabulary.
	DUP	Save a copy of the starting address.
	BEGIN	Go through the vocabulary.
	DUP SCAN	Find the next target definition.
	DUP	Keep a copy of this address.
	WHILE	
	TRIM	Re-link this target definition to target dictionary and

```
                                clean its header.
        REPEAT                  Repeat until the vocabulary is processed to the end. Link
                                Address of the last target word is left on the stack.
        2025 XOR dA @ -         Fix the link field of the last word in a vocabulary of the
        SWAP !                  target dictionary. If the link field is at 2025H, place a 0 in
                                it. Otherwise, put a 25H in it. This assures that the
                                FORTH and COMPILER vocabularies both end at the
                                first word #, the End-Of-Line definition.
        @ ,                     Fetch the link field address of the last word defined in a
                                vocabulary and store it on the top of the target dictionary
                                in RESET after FORGET where the initial vocabulary
                                link table is located.
        ;
: PRUNE ( -- )                  Re-link the target vocabulary and clean up all the headers
                                in the target definitions.
        {                       Switch to the target dictionary.
        CONTEXT 2 -             Head of the COMPILER vocabulary.
        DUP CLIP                Re-link the target COMPILER vocabulary.
        1+                      Head of the FORTH vocabulary.
        CLIP                    Re-link the target FORTH vocabulary.
        }                       Switch back to the host dictionary.
        20 0 2025 2!            Patch the name field and link field in the first target word
                                '#'. Change its name to an ASCII BL with 0 character
                                count. Change its link field to 0 as the end of the
                                dictionary.
        EMPTY                   Cut the target dictionary completely off the host
                                dictionary.
        ;
```

5.9.5. Target Compiler in Action

Screen 3 in the source listing shows how the target compiler is used to recompile cmForth itself. Let's go through it line by line to see how the target dictionary is built up to the point it can be tested in the host system and booted up in a target system.

```
EMPTY                   Throw away any garbage previously collected on the host
                        dictionary.
2 LOAD                  Load the target compiler.
HEX 2000 800 0 FILL     Clear the target dictionary for ROM code.
2000 H' !               Initialize the target dictionary pointer.

: BOOT  ( --)           Define BOOT and put its header in the host dictionary.
        }               Exchange the target dictionary pointer with the host dictionary
                        pointer. The following words will be compiled into the target
                        dictionary. It is the reset routine.
        16 FFF FOR 0    Copy 4K words from ROM to RAM.
```

```
        @+ 1 !+ NEXT
        -1 @            Pulse the clock input of the 74HC74 flip-flop to turn on RAM
                        and turn off ROM.
        ; ( reset)      This cell is later replaced by the reset routine 'reset' which
                        initializes NC4000 in the ForthKit system.

0 , 0 , 0 , 0 , 0 , 0 , 0, 0 , 0 , 0 ,      Initial values for local memory and some system variable.
0 ,
0 ,                                         TIB.
77C0 ,                                      This cell will be used for SPAN. However, 77C0 is the address of
                                            the terminal input buffer, which will be copied into TIB.
0  , 0 , 0 , 0 , 0 ,                        Initial values for other system variables: >IN, BLK, dA, ?CODE,
                                            and CURSOR.
1FF ,                                       SO
A ,                                         BASE
0 ,                                         H
DECIMAL 521 ,                               C/B for 5 MHz clock.

 {; interrupt }     ( -- )          Compile a noop interrupt service routine at 20H. The
                                    header is compiled in the host dictionary.
                    POP DROP        Discard the return address saved on the return stack by
                                    the interrupt. Execute the next word as if nothing had
                                    happened.
                    ;

0 , 0,                      Vocabulary link table for FORTH and COMPILER.
1 ,                         CONTEXT
```

In the target system, the code of BOOT routine from 2000H to 200FH will be copied to 1000H to 100FH in the target system, and the variable initialization table from 2010H up together with the target dictionary will be relocated to 1OH-7FFH in the target system.

Following the above variable initialization table, we will build the target dictionary by compiling cmForth source screens.

```
:#      R>DROP ;        This is the word EXIT. # is a temporary name. It will be changed
                        to 80A0H later to perform end-of-line function.

DECIMAL 7 11 THRU       Load nucleus.
12 22 THRU              Load the interpreter.
23 24 THRU              System initialization.
' reset dA @-           Install the reset vector at the end of BOOT.
HEX 2009 ! DECIMAL
25 30 THRU              Load compiler.

}                        At this point, the target system is complete. Restore host dictionary
```

for normal usage.

The newly compiled target dictionary image can be burned into EPROM's and inserted to the ROM sockets. If the hardware is done right, the new computer with NC4000 can now be powered up and it should say "hi" on your terminal.

However, before you burn the EPROM's, it would be nice if the new target system can be tested in the host system. The code at the bottom of Screen 3 does exactly that. The newer cmForth assumes that NC4000 boots up in a shadow ROM and operates in RAM, which can be overwritten with the core image of the target system. In the older cmForth, the testing process was more complicated because the target system must be run while in the area 2000H-27FFH. The code for testing was compiled differently than the code for the target system.

PRUNE		Re-link the target dictionary to produce a core image of a stand-alone cmForth system.
: GO	(--)	Copy the target core image in the area 2000H-27FFH to the low RAM memory area and pass control to the new system
	FLUSH	Clean up the disk buffers.
	[HEX)	
	2015 4 I!	Store source address in MD.
	15	Destination address.
	6EA FOR	Copying loop.
	4 I@!	Exchange source and destination addresses.
	1 @+	Fetch from source.
	4 I@!	Exchange addresses again.
	1 !+	Store to destination. NEXT
	2009 PUSH	Push 2009 on the return stack.
	;	Jump to 2009H and execute the 'reset' routine to bring up the new system.

Chapter 6. Programming Tips

The source code in cmForth is a large reservoir of programming tools and examples you can use to solve many practical problems. However, programming examples can never be too many. In this chapter, I will discuss a number of exercises that I have worked out on my NC4000 machine under cmForth. I tried to explore areas outside of the operating system which was addressed extensively by cmForth. I hope these exercises will be helpful for people who are more concerned with day-to-day programming problems than with the problems encountered by the operating system.

Chuck Moore provided many excellent examples using NC4000 to control many different peripheral devices as App Notes distributed with the ForthKits. These App Notes are also reproduced in *More on NC4000, Volumes 4 and 5*. Many NC4000 users also contributed code, applications, tips, and insights related to NC4000 in *More on NC4000*. Interested readers should consult these and other volumes for further information.

6.1. Benchmarks

Benchmarks do not lie. Liars do benchmarks.

I have to admit that I have an ax to grind, and it is always nice to show off your newest toy to friends and relatives. My favorite benchmarks are simple tests enclosed in big loops so that one can use a regular clock to time them. Some of these tests are shown in Figure 6.1.

The interesting thing about these test programs is that the blazing speed of NC4000 makes them difficult to time accurately if you only do 32768 loops. For example, it takes NC4000 about 8 ms to complete 32768 empty FOR-NEXT loops, about 100 times faster than my PC. To facilitate testing, I have to put the test program inside another loop.

The reason why NC4000 is faster than any conventional microprocessor is very simple. NC4000 executes one machine instruction per clock cycle while other microprocessors need more than one clock cycle to do anything. The number of clock cycles required to perform an equivalent task is a more useful measure of microprocessor's performance than benchmarks. Table 6.1 shows the comparison among NC4000, 68000, and 8086.

Because NC4000 does most operations in one machine clock cycle, in many instances several operations in one cycle, it is no wonder that it should be faster than 68000 and 8086 running at much higher clock frequencies. The most interesting instructions are subroutine calls and returns, which are optimized to the physical limit of computer design--one clock cycle for calling and zero cycle for returning. In running conventional high level language programs, subroutine calls and returns often consume as much as 40% of the execution time. By reducing the overhead of subroutine calls and returns to the bare minimum, NC4000 can support other high level languages and significantly improving their efficiency.

Table 6.1. Machine Cycles for 16 Bit Integer Operations

Operation	NC4000	68000	8086
Register-Register Move	1	8	2
Register-Memory Move	2	16	8-11
Register-Register Add	1	8	3
Multiply	23	74	118-133
Divide	31	144-162	144-162
Call Subroutine	1	32	19-28
Subroutine Return	<1	32	8-18
Push Register	<1	16	10
Pop Register <1 16 8 Branch	1	18	4-16

```
( BENCH MARKS, 23MAR85CHT ) HEX
: LOOPS   7FFF FOR NEXT ;
: LOOPTEST   FOR LOOPS NEXT ;
: +TEST   7FFF FOR I DUP + DROP NEXT ;
: +TESTS   FOR +TEST NEXT ;
: -TEST   7FFF FOR 7FFF I - DROP NEXT ;
: -TESTS   FOR -TEST NEXT ;
: *TEST   7FFF FOR I DUP * DROP NEXT ;
: *TESTS   FOR *TEST NEXT ;
: /TEST   7FFF FOR 7FFF I / DROP NEXT ;
: /TESTS   FOR /TEST NEXT ; DECIMAL
```

Figure 6.1. Sample Benchmark Programs.

6.2. WORDS--Listing the Vocabulary

VLIST in figForth and WORDS in F83 are very useful utilities to examine the status of the dictionary. These words are addictive. Once you get used to them, life seems impossible without them. However, the tradition of polyForth, which could be attributed to Chuck Moore, was not to bother with them. VLIST or WORDS were not useful in polyForth environment, because of the 8-way threading and hashing of the dictionary-vocabulary structure and the three character names. In cmForth, Chuck eliminated the 8-way hashing of the dictionary, retaining only two vocabularies. The dictionary structure is now simple enough so that VLIST or WORDS can be meaningful again. Since only the first three characters are retained in the name field of a word, you cannot completely recover the full name of a word. However, the mechanism to retain names up to 31 characters is built into the cmForth system. If you really want full names in your dictionary, you can do it by target compiling the system with the variable WIDTH set to 16.

In Figure 6.2, I have shown a simple implementation of WORDS which lists the word names in a vocabulary. In the listing, each name is preceded by its character count. You can improve it to make the listing prettier, or display the address alone with the name, etc.

cmForth has only two vocabularies, FORTH and COMPILER. In the version we have here, FORTH and COMPILER are not defined. Their definitions are trivial, and are shown in Figure 6.2. FORTH stores a 1 into the variable CONTEXT and COMPILER stores a 2 in it. This number in CONTEXT is to be subtracted from the address of CONTEXT to get the address of a pointer pointing to the link field of the last definition in one of the two vocabularies. The phrase in the definition of WORDS:

CONTEXT DUP @ - C

thus obtains the link field address of the last definition in the context vocabulary.

'C.' is similar to EMIT, except it will not emit non-printable characters. It is useful in doing ASCII dumps. It was needed when I first coded the ID routine which displayed three characters in the name field even though the name contains only one or two characters. The revised ID shown in Figure 6.2 is smart enough to display one or two character names correctly, and C. is not necessary anymore. '.ID' takes a link field address, prints the content of the name field, and returns with the link field address of the next word in the same vocabulary. With this rather powerful name printing word, the definition of WORDS becomes very straightforward. WORDS follow the context vocabulary and prints the names of all the words in this vocabulary. It stops when a link field address is 0, which indicates the end of this vocabulary.

```
( WORDS, 23MAR86CHT )
: FORTH   1 CONTEXT ! ;
: COMPILER   2 CONTEXT ! ;
: C.   >R   31 127 I WITHIN
   IF R> EMIT   ELSE R> DROP THEN ;
: .ID   ( A - A' )
   1 @+ 2C@+ 31 AND DUP >R . C.
   2C@+ I 1 > IF C. ELSE DROP THEN
  R> 2 >   IF C. ELSE DROP THEN   3 SPACES   DROP ;
: WORDS   CONTEXT DUP @ -
   BEGIN  .ID   DUP 0= UNTIL   DROP ;
```

Figure 6.2. Vocabulary Definitions and WORDS.

6.3. Memory Dump

A good memory dump utility is always handy when you have to do detective work in the object code. The DUMP word in cmForth is unconventional. It takes an address from the stack, displays the contents of 8 cells starting from this address, and returns with the address of the cell after the last cell displayed-. The reason why Chuck coded it this way was that he was experimenting with a CRT display circuit driven directly by NC4000 chip. "In this display, he used only the top line for command entry and the second line for responses from NC4000. In this scheme, he could display only one line at a time. That's why his DUMP dumps only one line of data. Because DUMP returns the address of the next line, DUMP can be used repeatedly to scan a section of memory.

Since you are more likely to use a regular 24 line CRT display terminal or a computer as terminal/disk server, a multiple line DUMP routine would seem to be more useful. Figure 6.3 shows such an implementation.

CHAR displays only printable characters. It substitutes a blank for any non-printable character. The most significant bit of the character is striped. TYPE takes a cell address and a byte count as arguments, and displays a string of characters. It assumes that the byte count is always even.

(DUMP) is similar to the DUMP in cmForth. It dumps only one line or 8 cells of memory and bumps the address by 8. The difference is that (DUMP) does not include carriage return and line feed. DUMP calls TYPE and (DUMP) alternately, and displays a nicely formatted dump on the terminal.

```
( MEMORY DUMP, 23MAR86CHT )
: CHAR ( C )   127 AND 32 MAX EMIT ;
: TYPE ( A #)   2/ 1 - FOR   2CQ+ CHAR CHAR   NEXT DROP ;
: (DUMP) ( A - A' )
   DUP 5 U.R SPACE
   7 FOR   1@+   SWAP 5 U.R   NEXT   ;
: DUMP ( A # )
   8 / 1 -   FOR
   CR   DUP 16 TYPE   3 SPACES   (DUMP)   NEXT   DROP ;
```

Figure 6.3. Regular DUMP Routine.

6.4. Line Editor

In my system, the source code is entered and edited inside the IBM PC using F83 editor. It is a convenient environment to write and change source code. However, it would be nice to do the editing directly inside NC4000 without having to switch back and forth between F83 system and NC4000. A small line editor is shown in Figure 6.4. Screen 16 in Figure 6.4 contains the basic functions for listing a screen of source code in the buffer memory of NC4000.. Screen 17 has a few of the elementary line editing commands.

Because NC4000 can process 16 bit numbers much faster than 8 bit bytes, screens of source code are stored in 1024 cell disk buffers. The text string parser in cmForth also assumes that the input character stream is cell based, not byte based. To display one line (64 characters) of code, it is very convenient to use the incremental fetch instruction @+ in a simple loop to obtain the character string and display it. That's what LINE does, given the screen number and a line number on the stack. T or the type command is a simple derivative of LINE.

LIST is defined to repeat LINE 16 times with a little extra formatting to boost the screen image. L uses the content of variable SCR to do the listing.

Only the commonly used line editing commands are defined in Screen 17. The cornerstone is the command P, which accepts a character string from the terminal and copies it to the current line in the current screen being edited. I encountered a few problems in debugging these editing commands. One problem is that the input character string obtained by WORD is stored in the word buffer as a byte string. This byte string had to be converted into a cell string before moving into the disk buffer. Another problem is the MOVE command, which has a very strange behavior: it copies the source cells in the forward direction and stores them to the target memory in the backward direction. After much grief, I saw the light, and thereafter coding was rather straightforward.

Another feature in cmForth concerning the disk buffer is that the most significant byte in each cell is assumed to have 40H in it. When you fill the buffer with blanks, you have to write 4020H into each cell. When you search for the " character in the input stream, the pattern given to WORD must be 4094H instead of 94H, the ASCII code of ".

These line editing commands can accomplish quite a bit of editing. To do more precise and efficient editing, one would probably need a good string editor, too. The string editor is left as an exercise for the reader.

```
Scr # 16   B:NC4000.BLK
0 ( LIST, 23MAR86CHT )
1 VARIABLE SCR   VARIABLE L#
2 : LINE ( SCR # )
3   64 * SWAP   BLOCK +
4   63 FOR   1 @+ SWAP   EMIT   NEXT   DROP ;
5 : LIST   ( SCR )
6   DUP SCR !   DUP CR ." SCR# ".
7   0   15 FOR   2DUP CR   DUP 3 U.R 2 SPACES
8   .LINE   1 +   NEXT   2DROP ;
9 : T ( N)   DUP L# !   CR   SCR @ SWAP .LINE   ;
10 : L   SCR @   LIST ;
11
12
13
14
15


Scr # 17   B:NC4000.BLK
0 ( LINE EDITING, 23MAR86CHT )
1 64 CONSTANT C/L   HEX
2 : WHERE ( N )   C/L *   SCR @ BLOCK +   ;
3 : P   L# ~ WHERE   DUP C/L 4020 FILL   4094 WORD   2* DUP C@
4   BEGIN   DUP WHILE 1 - >R
5      1 + DUP C@   4000 +   ROT 1 !+   SWAP R>
6   REPEAT DROP 2DROP   UPDATE ;
7: M( N M)   1 + WHERE 1 - SWAP WHERE SWAP
8   C/L 1 - MOVE   UPDATE ;
9: U   L# @ 1 +   OF OVER -
10   FOR   DUP I + DUP 1- SWAP M NEXT DROP   P ;
11 : X   L# @ 1 +   OF OVER -
12   FOR DUP DUP 1 - M   1 +   NEXT   DROP
13   OF WHERE C/L 4020 FILL   ;
14 DECIMAL
15
```

Figure 6.4. Line Editor

6.5. Stack Pictures

It is always nice to know your stacks. In most instances, you have to be sure of the items you've pushed on the data stack before embarking on to your next task. When you test and debug a word, it is very helpful to have a utility word which displays the contents of the data stack non-destructively. The word S in many Forth system is very popular for this reason. With NC4000 and cmForth, the problem is that the stacks do not have bottoms or tops! The external stacks have 256 cell capacity because NC4000 provides 8 bit stack pointers. The stack pointers are incremented or decremented in modulo 256 and the stacks wrap around and fold into themselves. Without knowledge of where the data stack begins or ends, it is impossible to define .S in the normal sense.

My proposition as shown in Figure 6.5 is to display only the top 5 elements non-destructively on the data stack. To get to the fifth element, I move the first four elements to the return stack. Then these five elements are duplicated, printed, and restored back onto the data stack.

The command .RS does the same thing to the return stack. Its usefulness is limited. When you execute .RS from the keyboard, the picture of the return stack is always the same. It is intended to be used inside nested definitions to show the nested return addresses.

```
( STACK PICTURE, 24MAR86CHT )

:.S ( DISPLAY TOP 5 STACK ELEMENTS )
   >R >R >R >R
    DUP .    R> DUP .    R> DUP .    R> DUP .    R> DUP .    ;

:.RS ( DISPLAY TOP 5 RETURN STACK ELEMENTS )
    R> R> R> R> R>
    DUP . >R    DUP . >R    DUP . >R
    DUP . >R    DUP . >R    ;
```

Figure 6.5. .S and .RS to Show Stack Pictures.

6.6. Display Internal Registers

There are 17 addressable internal registers inside NC4000, which indicate the current status of the machine. Generally, detailed knowledge about these internal registers is not required to use NC4000 machine. Occasionally, one might want to know their contents for debugging purposes.

The command I defined in Figure 6.6 displays all the accessible internal registers in a nicely formatted fashion. The data stack pointer and the return stack pointer are isolated from the J/K register and displayed separately. The most useful information you can get from this display is the content of the data stack pointer and the registers controlling the B and X ports. The main purpose of I is to satisfy your curiosity on the inner mechanism in NC4000.

```
( INTERNAL REGISTERS, 24MAR86CHT )
: (.I) ( N1 N2 N3 N4 )
   CR   3 FOR 8 U.R NEXT ;
: .I ( DISPLAY NC4000 REGISTERS )
   CR ."         J         K         I         P"
    2 I@    1 I@    0 I@ 256 /MOD    (.I)
   CR .”  MD        SRI”
    7 I@    6 I@    5 I@    4 I@    (.I)
   CR ."    B-PORT     MASK     I/O TRISTATE"
   11 I@    10 I@    9 I@    8 I@    (.I)
   CR ."    X-PORT     MASK     I/O TRISTATE"
    15 I@    14 I@    13 I@    12 I@    (.I)    ;
```

Figure 6.6. Internal Registers.

6.7. Input and Output

Among the 17 internal registers, 8 are devoted to control the two I/O ports: the 16 bit B-port and the 5 bit X-port. The high percentage of resources in NC4000 allocated for I/O ports reveals Chuck Moore's intent for NC4000. It is optimized to be a super-fast controller. In the prototype version of NC4000 using the 3 micron CMOS technology, each I/O pin can source or sink 16 mA of current. This large driving capability makes it very easy to use NC4000 to drive other electronic devices without additional buffering or amplifying chips.

Here I wish to demonstrate how these ports may be used to do simple I/O tasks. It was often said in the microprocessor business: "If you can turn a LED on and off, you can do anything."

When NC4000 is powered up after RESET, the B-port is initialized to be a 16 bit output port and the output pins are all pulled to ground. The X0 pin is configured to be the transmitter of the terminal interface, and X4 the receiver. If your NC4000 is using the serial terminal interface to talk to your terminal or your PC, be careful and don't bother the X-port until you know exactly what you are doing. Accessing X-port might sever the serial interface to the terminal and you might have to reset NC4000 to bring it back. B-port is free for you to experiment.

Figure 6.7 shows the commands INPUT and OUTPUT. INPUT configures the B-port as a 16 bit input device, reads the data from BO-B15 pins, and returns it on the stack. OUTPUT configures it to be a 16 bit output port and sends the top of stack item to the pins BO-B15. You can connect an oscilloscope probe to any B-port pin and execute the OUTPUT command to drive the scope trace up or down. You can also connect a LED lamp between a B-port pin and ground to see if you can turn the LED on and off. To test the INPUT command, you will have to connect a switch between a B-port pin and ground. Remember also that you have to pull the B-port pin to 5 volt through a resister. Then you can execute INPUT to read the status of the switch.

Any microprocessor can be programmed to turn LED's on and off. The advantage in using NC4000 is that it can do these things faster than any other microprocessor. In fact, what I would like to demonstrate is programming NC4000 to do the on-off switching at 4 MHz, the speed of the clock driving NC4000. The code is shown in Figure 6.7, Screen 21.

The only way to output data at the clock rate to the B-port is to pop data from the data stack and copy it to the data register in the B-port. Obtaining data from main memory would take two cycles. Re-calculating data and sending the results to B-port would also take at least two cycles. If the data is stored on the data stack, it can be popped to the B-port in a single cycle. The command ZEROS pushes n 0's on the data stack, and ONES pushes n-1's on it. FLIP-FLOPS pushes a number of 0 and -1 pairs on the stack. If you fill the entire data stack with zeros and -1's, you can then pop these words out to the B-port indefinitely at the rate of 4 million words per second. FAST dumps 256 words and FAST-DEMO repeats FAST in a FOR-NEXT loop.

If the data stack is filled with alternate ones and -1's, FAST-DEMO creates a square wave at all 16 B-port output pins at a frequency of 2 MHz. Using ONES and ZEROS one can generate different square wave patterns as output. In this manner, you have a set of 16 programmable flip-flops running at 4 MHz. There might be cheaper ways to build flip-flops, but this is the only

microprocessor which can simulate flip-flops at this speed.

```
Scr # 20    B:NC4000.BLK
0 ( I/O DEMO, 24MAR86CHT )
1 : INPUT ( -- N, READ A 16 BIT NUMBER FROM B-PORT)
2   0 9 I! ( MASK )    0 10 I! ( DIRECTION )
3   0 11 I! ( TRISTATE )
4   9 I@ ( INPUT DATA )    ;
5 : OUPUT ( N --, SEND N TO B-PORT )
6   0 9 I! ( MASK )    -1 10 I! ( DIRECTION )
7   0 11 I! ( TRISTATE )
8   8 I! ( OUTPUT DATA )    ;
9
10
11
12
13
14
15


Scr # 21   B:NC4000.BLK
0 ( 4 MHZ PROGRAMMABLE FLIP-FLOP, 24MAR86CHT )
1 : ZEROS ( N --, PUSH N ZEROS ON THE STACK )
2   FOR   0   NEXT   ;
3 : ONES ( N --, PUSH N -1'S ON THE STACK )
4   FOR   -1   NEXT   ;
5 : FLIP-FLOPS ( PUSH ALTERNATE ONES AND ZEROS ON STACK )
6   129 FOR   0 -1   NEXT   ;
7 : FAST ( PUMP 256 WORDS FROM STACK TO B-PORT )
8   256 TIMES 8 I! ;
9 : FAST-DEMO ( N --, DO FAST N TIMES )
10    FOR   FAST   NEXT   ;
11 ( FLIP-FLOP    -1 FAST-DEMO )
12  ( SHOW 4 SECONDS OF A 2 MHZ SQUARE WAVE ON ALL B-PORT PINS.)
13
14
15
```

Figure 6.7. Input and Output Demonstrations

6.8. PICK and ROLL

Thou shalt not PICK; and Thou shalt never ROLL.

That was the advice attributed to Chuck Moore. If you modularize your words properly, you should never need to access the stack below the third element. Thus DUP, SWAP, OVER, and ROT should suffice, and these are words you get in cmForth. If you find yourself in a situation in which you have to access items below the third item on the data stack, it's time to rethink your algorithm.

Nevertheless, Forth 83-Standard Team saw new light and insisted that PICK and ROLL be included in the standard to make room for fuzzy thinking and lazy programming. Since PICK and ROLL are in the standard, you might just as well do it as programming examples.

Figure 6.8 shows the definitions of PICK and ROLL. They are very similar. To gain access to the nth element on the stack, I tuck the first n-1 elements under the top element on the return stack. Because I am using a FOR-NEXT loop to move these elements, the top element on the return stack, which is the loop counter, must be preserved. After moving the n-1 elements out of the way, I can either duplicate the nth element into the MD register for PICKing, or move it to MD for ROLLing. The stacks are then restored to the correct state and the content in MD is finally pushed back on the data stack.

```
( PICK AND ROLL, 28MAR86CHT )
: PICK ( N - N' )
   DUP 6 I! ( SAVE N IN SR )
   ?DUP IF 1- FOR R> SWAP >R >R NEXT THEN ( MOVE TOP ELEMENTS )
   DUP   4 I! ( PICK IT TO MD )
   6 I@ ( RETRIEVE N )
   ?DUP   IF 1 - FOR   R> R> SWAP >R NEXT THEN ( RESTORE )
   4 I@ ( GET NTH ITEM BACK )   ;
: ROLL ( N - )
   DUP 6 I! ( SAVE N IN SR )
   ?DUP IF 1- FOR R> SWAP >R >R NEXT THEN ( MOVE TOP ELEMENTS )
   4 I! ( ROLL IT TO MD )
   6 I@ ( RETRIEVE N )
   ?DUP   IF 1 - FOR   R> R> SWAP >R NEXT THEN ( RESTORE )
   4 I@ ( GET NTH ITEM BACK )   ;
```

Figure 6.8. PICK and ROLL.

This is one of many possible ways to implement PICK and ROLL. It is not very fast because of the thrashing activity on the return stack. If you have a scratch pad of 256 cells somewhere in the main memory, moving n-1 elements can be done much faster using ~+ and !- instructions. You might want to try it for yourself.

Including PICK and ROLL here does not imply that I approve of their use.

6.9. Square-Root

A very unique instruction in NC4000 is the square-root step S'. By repeating this instruction 16 times, you can take the square root of a double integer very easily and very quickly. Chuck Moore included this feature in NC4000 because he wanted to use NC4000 to do fast graphic processing, and he needed the square-root function frequently. The square-root step is very similar to the divide step /'. It does a conditional subtraction; and if overflow condition occurs, the result of subtraction is not written back to the T register.

The square-root routine is shown in Figure 6.9. It takes a positive double integer on data stack as input and returns a positive integer as square root.

Because of problems in handling the carry condition, the prototype NC4000 chip cannot take the square-root of numbers greater than 16K.

```
( SQUARE ROOT, 29MAR86CHT )
: SQRT ( D -- N )
   32768 6 I! ( SR REGISTER ) 0 4 I!   ( MD REGISTER )
   D2*   14 TIMES S'    ( SQRT STEPS )
   DROP   ;
```

Figure 6.9. Square Root.

6.10. Terminal and Disk Server on IBM-PC

I am using an IBM-PC computer as the host of NC4000 machine. NC4000 talks to the PC through the 9600 baud COM1 serial channel and uses the serial disk protocol in cmForth to access the floppy disks in the PC. The host interface is programmed using F83 Forth system. Files are opened and managed by F83. Source code in the files are entered and edited with the F83 editor. The serial disk at 9600 baud is slow, but adequate for my purposes.

I did some experiments with PC-DOS and even the BIOS. Somehow, PC always manages to lose characters if the COM1 port is read through DOS calls or BIOS service interrupt. The PC spends so much time playing with the characters that it just cannot get the characters and put them into either the disk buffer or display them on the screen reliably, even though it takes a whole millisecond for a character to get through the COM1 serial port.

The code presented in Figure 6.10 tries to manage the COM1 port and to make the PC to serve NC4000 faithfully. As a terminal server, the PC loops on the COM1 receiver. Whenever a character is received, it is sent directly to the CRT display buffer. The display is not scrolled. When the last line is displayed and a carriage return is detected, the next line will be displayed at the top of the screen. When an ASCII NUL is received, the disk server will be invoked to handle the sending or receiving of text block to or from NC4000.

This server works very reliably, with one exception. If NC4000 sends lots of characters without carriage returns wisely dispersed in the character stream so that the characters overflow the CRT screen buffer, the terminal server will get lost. You will have to use ESC to return to F83 and re-establish communication using the NC command.

```
Scr # 1   B:NC4000.BLK
0 \ Com1 and Com2   12dec85cht
1 HEX
2 B800 CONSTANT SCREEN
3 3FD CONSTANT STAT 3F8 CONSTANT DATA 4 DECIMAL   2 5 THRU 5 EXIT
6 Characters obtained from NC4000 are put into the screen buffer
7 directly. 3F8 is the data register in COM1 8251 and 3FD is the
8 status register.
9 COM1 must be initialized by the DOS command:
10   >MODE COM1:9600,n,8,1
11 to set up the baud rate and character format.
12 In F83,   OPEN NC4000.BLK   OK load in this program.
13 Type   NC to connect to NC4000 board.
14 While NC4000 is the master, pressing ESC key returns you back
15 to F83.


Scr # 3   B:NC4000.BLK
0 \ Chip
1 CREATE I/O HEX 400 ALLOT ASSEMBLER
2 LABEL EOL AO # BL MOV DI AX MOV BL DIV BL CL MOV
3   AH CL SUB CH CH SUB AL AH MOV AL AL SUB REP AL STOS
4   18 # AH CMP   0= IF DI DI SUB THEN   RET
5 LABEL RCV   STAT # DX MOV   BEGIN 0 AL IN 1# AL AND 0<> UNTIL
6   DATA # DX MOV 0 AL IN RET
7 CODE XMT   STAT # DX MOV   BEGIN 0 AL IN 40 # AL AND 0<> UNTIL
8   AX POP DATA # DX MOV 0 AL OUT NEXT END-CODE
9 LABEL RECEIVE   400 # CX MOV   I/O # BX MOV   BEGIN
10   CX CX OR   0<> WHILE   RCV #) CALL   AL 0 [BX] MOV
11   BX INC   CX DEC   REPEAT   RET
12 : BLOCK-XMT   BLOCK   400 0 DO   DUP C@ XMT   1+ LOOP DROP ;
13 : DISK   DUP 0< IF   ( RECEIVE )   I/O SWAP 7FFF AND BUFFER
14   400 CMOVE UPDATE   ( 0 XMT)   ELSE BLOCK-XMT   THEN   ;
15 DECIMAL



Scr # 5   B:f1C4000.BLK
0 \ Chip   12DEC85CH'I
1 CODE NC   HEX   ES PUSH   SCREEN # AX MOV   AX ES MOV
2   CLD   CH CH SUB   AO # BX MOV   DI DI SUB   BEGIN
3   STAT # DX MOV   0 AL IN   1 # AL AND
4   0<> IF   DATA # DX MOV   0 AL IN
5      AL AL OR   0= IF RCV #) CALL   AL AH MOV
6         RCV #) CALL   ES PUSH   DI PUSH   AX PUSH- AX AX OR
7         0< IF   RECEIVE #) CALL   THEN   C: DISK ;C   DI POP
8 ES POP   ELSE   OD # AL CMP   0= IF EOL #) CALL
9           ELSE   7 # AH MOV   AX STOS   THEN
1  0   OF5E # AX MOV   AX STOS   DI DEC DI DEC   THEN
11  THEN   100 # AX MOV   16 INT
12   0<> IF   AX AX SUB   16 INT   AH AH SUB
13      1B # AL CMP   0= IF   ES POP   NEXT THEN
14      DATA # DX MOV   0 AL OUT   THEN
15   AGAIN   END-CODE   DECIMAL
```

Figure 6.10. Terminal and Disk Server

```
Scr # 2   B:NC4000.BLK
0\ Call high level words   12dec85cht
1 ASSEMBLER
2 LABEL HILEVEL
3   RP DEC RP DEC   IP 0 [RP] MOV   IP POP   NEXT
4 : C:
5   [ ASSEMBLER ]   HILEVEL #) CALL   FORTH   ]   ;
6 CODE (;C)
7   IP PUSH   0 [RP] IP MOV   RP INC   RP INC
8   RET END-CODE
9 : ;C  [ ASSEMBLER ]   COMPILE (;C)   ASSEMBLER
10   [COMPILE]   [   ;   IMMEDIATE
11 EXIT
12 Henry Laxen's trick to allow assembly routine to call high
13 level colon words.
14
15


Scr # 4   B:NC4000.BLK
0 \S Chip   12dec85cht
1 I/O   1K buffer to receive block data from NC4000
2 EOL Subroutine to process carriage returns from NC4000.
3
4
5 RCV Subroutine to grab one character from NC4000.
6
7 XMT Code word to transmit one character to NC4000.
8
9 RECEIVE Subroutine to receive one block of characters from
10   NC4000.
11
12 BLOCK-XMT Transmit one block of characters to NC4000.
13 DISK   The disk service routine. The serial disk.
14
15

Scr # 6   B:NC4000.BLK
0 \S Chip   12dec85c!;t
1 NC   The interface between NC4000 and PC through COM1.
2   Initialize screen buffer pointers.
3   Begin
4      If a character is received from NC4000,
5         If the character is a NUL, do disk service.
6         If the character is a CR, do End-of-Line service.
7         If it is a regular character, store it in screen
8            buffer and bump pointer.
9      Else
10         If a character is received from the keyboard,
11            If the character is a ESC, return to F83.
12            Else send it to NC4000.
13         Then
14       Then
15   Again
```

Figure 6.10. Terminal and Disk Server (cont'd)

6.11. Arcsine by Interpolation

There are many occasions in which you have to evaluate a rather complicated function which is not very easy to compute, particularly with a 16 bit integer machine or Forth. If high accuracy is not required, it is very easy to get an answer by interpolation among an array of known points. I encountered a situation that I had to compute arcsine function, converting sine and cosine values to angles in degrees. I was allowed to trade accuracy for speed, because the angles are used only for refreshing a numeric display for an operator to make sure that the system is functioning.

Interpolation is extremely simple in Forth using the ratio operator */, as shown in the source code in Figure 6.11. The accuracy depends upon how large a data table is allowed for interpolation. In our case, we used a 20 point table to represent angles from 0 to 90 degrees. It is easy to extend this table for more accurate interpolation.

In Figure 6.11, the arcsine table is defined as (ARCSIN). The entries in this table are in the units of 0.1 mili-radians, from 0 to PI/2 (15708 as the last entry.) The input to the interpolation function ARCSIN is the sine of an angle, multiplied by a scaling factor of 10000, and the output is an angle in degrees multiplied by a scale factor of 100. The absolute value of sine is divided by 500, 20th of the range 10000, with both quotient and remainder retained. The quotient is used to retrieve a pair of neighboring values in the (ARCSIN) table and the remainder is used to compute the exact position between these two neighboring points. The resulting angle in radian is then converted to degrees with the sign restored.

This method can be used to approximate any complicated function which does not render itself easily to integer arithmetic. You only have to supply a table of function values. The size of the table can be optimized according to the required accuracy of approximation. The computation involves only a /MOD, a table look-up, and a */ . I used an extra */ to scale the output. It is very fast and does not depend upon the complexity of the function.

```
 ( interpolation 16aug86cht )
CREATE (ARCSIN) ( a table of function values )
0 , 500 , 1002 , 1506 , 2014 , 2526 , 3046 , 3576 , 4116 ,
4668 , 5240 , 5824 , 6434 , 7076 , 7754 , 8480 ,
9272 , 10160 , 11198 , 12532 , 15708 ,
: ARCSIN ( 10000*SIN -- 100*ARCSIN )
   DUP >R   ABS
   10000 MIN   500 /MOD ( 2* )   (ARCSIN) + 2@   DUP >R   -
   500 */   R> +   9000 15708 */
  R>   0< IF NEGATE THEN   ;
: ANGLE ( FRACTION BUCKET -- ANGLE*100 )
   10 - 1000 * +   10000 RADIUS @ */   ARCSIN   ;
```

Figure 6.11. Source code of interpolation.

6.12. High Speed Pattern Generator

NC4000 is a very fast machine, capable of executing one instruction every clock cycle. During one cycle, it can output one word to the B port, while doing several other tasks simultaneously. As discussed in Section 6.7 on I/O, you can program NC4000 to generate patterns at its clock rate, 4 MHz or more. The problem is to provide data stream to the B port so that large amount of data can be pumped out at this peak rate. To output data at the clock rate, data has to be pushed on the data stack, because it takes one cycle to pop a 16 bit number off the data stack and send it to the B port. To retrieve data from main memory and send out to the B port, at least three machine cycles are needed--two cycles of memory fetch and one cycle for output.

The data stack is only 258 words deep, which is not enough to make a usable pattern generator out of NC4000. Extending the data stack using bank switching or by extending the width of the data stack pointer to 16 bits, as I did in the design of the OF5493, does not solve the problem either because it is still very difficult to access the data stack randomly to retrieve different patterns.

A pattern generator must have the following properties to render it practical:

- It must be fast. 4 MHz is marginally acceptable. 2 or 1 MHz is becoming less interesting.
- It must be able to hold long sequences of patterns. Number of words in a pattern could be in the thousands or more.
- Patterns must be selected easily. Looping and sequencing through a number of patterns should be allowed.

Clearly, the B port in NC4000 does not meeting these criteria. An interesting alternative is using the main memory to store the patterns and to output the patterns directly.

This type of pattern generator is very useful in wave synthesis, digital signature source for device characterization. One particular application I had in mind is a microcode sequencer, which can be programmed to operate and test bit-slice microprocessor or microcontroller. In this application, a conventional sequencer is expensive and also difficult to program. A sequencer built around NC4000 would be easy to program because of the Forth underneath the system. Microcode can be deposit into the main memory and clocked out to operate the bit-slice machine. NC4000 is much more powerful than a sequencer because it can do loops and subroutines, nested almost indefinitely--every capability of high level language programming.

For bit-slice applications, the 16 bit word size in NC4000 is a serious limiting factor, because the sequencer generally requires many more bits to control the bit-slice engine. The width of patterns must be widened to 32 bits or more. In this pattern generator, I implemented 32 bit pattern width. It is easy to extend the width beyond 32 bits.

Using NC4000 to realize this pattern generator, you have to make use of two important features of NC4000: one is that you can use the upper 32K word data space to store the patterns or microcode and this data space can accommodate many 32K by 16 bit memory banks to provide enough width for desired microcode; and the other is that NC4000 can generate consecutive addresses at 4 Mhz clock rate using the following instruction phrase:

n TIMES 1 @+

given an initial memory address in the T register. One problem with @+ instruction is that the data in that memory location will be fetched into the N register and the original content of N register will be pushed on to the external data stack. For all intentions and purposes, we should assume that the data stack will be destroyed. Do not expect that anything you saved on the data stack can be retrieved later. If you really wanted to use the stack to pass parameters while generating patterns, you have to clean up the stack by SWAP DROP or NIP. Then, you have to put them in a FOR-NEXT loop with 1@+, costing many more cycles to output one pattern data.

Another interesting feature is that the memory address can be incremented or decremented by any integer from 1 to 31. This is a convenient way to double or triple the frequency of the output pattern or waveform. This is especially important in synthesizing musical notes, because once the waveform is stored in the microcode memory, one will get all the overtones for free.

The circuit schematic is shown in Figure 6.12.

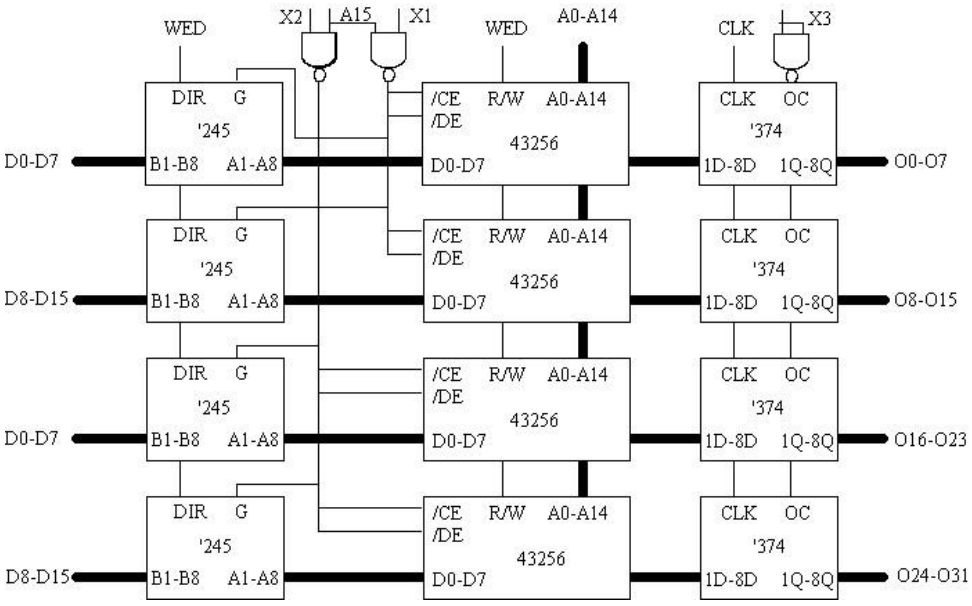


Figure 6.12. Schematic of the Pattern Generator.

The microcode memory consists of 4 uPD43256 32K by8 bit SRAM memory chips, divided into two 32K by 16 bit banks. Address lines AO-14 from NC4000 drives the respective address pins on these memory chips, and A15 is used to enable them. The data pins on these chips are connected to the data lines DO-15 through four 74HCT245 bus drivers. The two banks of memory chips are enabled together with their respective bus drivers by X1 and X2 line from NC4000. The microcode data are latched and sent to the external bit-slice engine through 4 74HCT374 latches which are connected to the data buses from the memory chips. The latches are enabled by X3 from NC4000, and clocked by the master clock which provides timing signal to NC4000.

To use the microcode memory, it has to be filled with known patterns from NC4000 host. This is done by first raising X1 alone and writing the top 16 bits of patterns into the first data memory bank. Then X1 is cleared and X2 is raised. Now the lower 16 bits of patterns can be written into the second bank of data memory. After microcode patterns are loaded into the data memory banks, the patterns can be clocked out through the '374 latches by raising all three X port lines X1, X2, and X3. Whenever NC4000 reads a memory location in the upper 32K data space, a 32 bit pattern stored in the RAM chips is put on the data bus and latched into the '374's. Using the program shown in Figure 6.13, you can generate a ramp function on the 32 output lines. You can use a scope to see that each line is switching at a different frequency.

```
( PATTERN GENERATOR, 17JUL87CHT )
OCTAL
: SEL ( n-- )   17 13 I! ( disable x0 )
   12 I! ( write n to x port ) ;
: RAMP   100000   77777 FOR   DUP DUP !   1 + NEXT   DROP ;
: 1TEST ( addr # -- garbage )   TIMES 1 @+   ;
: 2TEST ( addr #-- )   FOR   1 C~+ SWAP DROP   NEXT DROP ;
: 3TEST ( addr # -- )   FOR   1 !+ [ 100020 , ]   NEXT DROP ;
DECIMAL
```

Figure 6.13. Program to Control the Pattern Generator.

You might ask what happens on the data bus connecting to the NC4000, where the four '245's are also sending the 32 bit pattern to the 16 bit data bus. Well, I was told that this is a big no-no, because '245's are driving each other and eventually some weaker ones will be burnt out. So far, the 245's are working fine. It is probably better to use lines from the B port to enable the 32K RAM chips and the '245's separately. This way when you are outputting patterns to the '374's, the '245's can be disabled so that NC4000 data bus is isolated from the RAM data bus. You will then need 5 B port lines for total control over this pattern generator: one for '374's, two for the RAM's, and two for the '245's.

Some of the very elementary code to operate this pattern generator is shown in Figure 6.13. SEL enables one or more of the data bus drives and data latches. For example, 2 SEL enables writing the RAM's in Bank 1, 4 SEL does that for Bank 2, and 14 SEL enables the RAM's and the output latches so that 32 bit patterns are generated and sent through the latches. RAMP writes a ramp function into the enabled bank of RAM's for testing purposes. 1TEST is the program to send a sequences of patterns out from a memory area, but the stack is trashed in the process. To maintain a clean stack, 2TEST and 3TEST can be used. However, 2TEST and 3TEST takes longer to generate patterns because of the necessary NIP stack operation.

A sample test sequence is:

```
OCTAL
2 SEL RAMP                          ( initialize bank 1)
4 SEL RAMP                          ( initialize bank 2)
16 SEL                              ( enable RAM's and output latches) 100000
1000 1TEST
120000 7777 2TEST
160000 10000 3TEST
```

In conclusion, this pattern generator proves that NC4000 can be used to generate arbitrary digital data patterns at its clock rate. It is very useful in generating digital signatures and analog wave forms. Music synthesizer may be a good application. Programmable waveform generator is another. Bit-slice sequencer based on NC4000 is much cheaper than the one based upon the conventional sequencer design and much more versatile.

6.13. A/D Conversion with NC4000

I have always maintained that microprocessors are not computers. They are controllers. Among controllers, NC4000 is the fastest. There are lots of tasks where information and control functions are binary, like limit switches and power switches. However, real world information generally are presented in analog form. To obtain these information and respond to them bya microprocessor, analog-to-digital (A/D) converters are needed to digitize the analog signal so that the information can be stored and processed bythe microprocessor. It is much more fun to use the microprocessor connecting to the real world then to do abstract computation and simulation on a real computer.

I had an opportunity to design a system which measures the phase difference between two audio input channels such as in a stereo system. From the phase difference, one can infer the direction of the sound source. This is interesting because, in a sense, it emulates the hearing system of a human being. With two ears, we can quickly and accurately determine the direction of a sound source. I am not sure how the ears do this trick, but something like a correlation analysis should suffice.

I tried two different A/D converters and used two different approaches to integrate the A/D converters into an NC4000 system. The schematic diagrams of these two designs are shown in Figures 6.14 and 6.15.

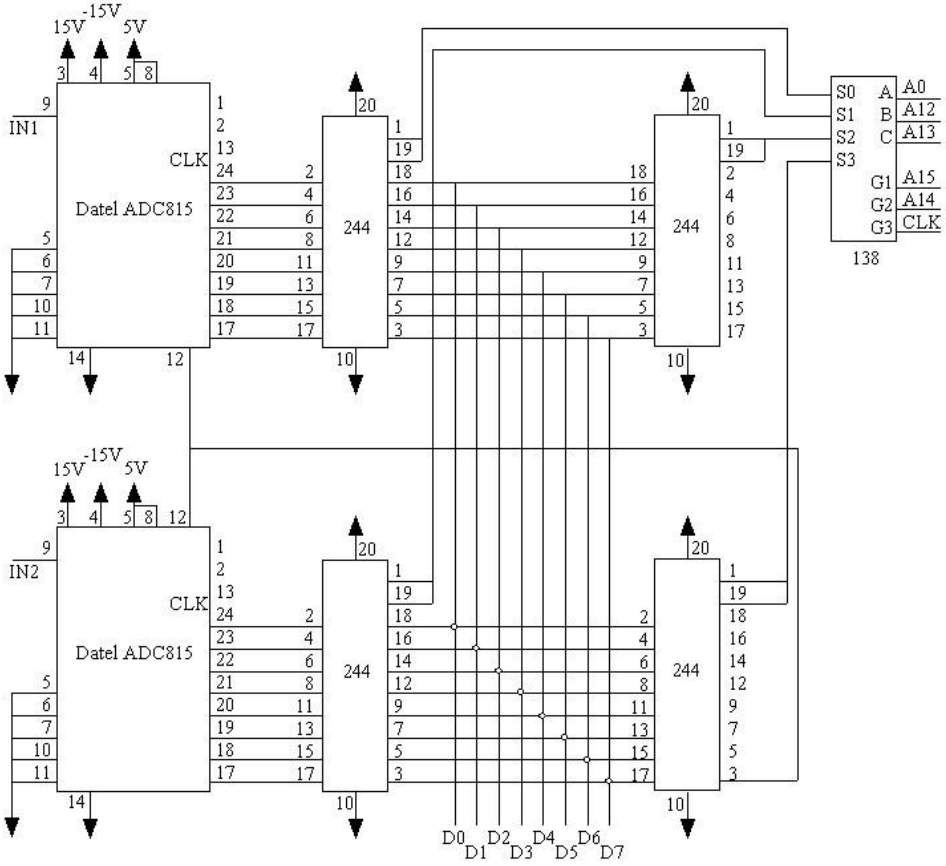


Figure 6.14. A/D Conversion with Datel ADC815.

Figure 6.14 shows two Datel ADC815 A/D converters hooked onto the memory bus of an NC4000 system. They are thus 'memory mapped' to two memory cells at 8000H and 8001H. The A/D is only 8 bits wide, and the data is fed to the lower 8 bits of the memory data bus through a pair of 74HCT244's. The A/D chip needs a strobe pulse to start a conversion cycle. This strobe is provided by writing a -1 to the memory location 9001H, also through a 74HCT244. A 74HCT138 decodes the address lines and enables the '244's.

The code to control the A/D pair and grab a block of data from these A/D converters is shown in the first screen in Figure 4.16. 120 pairs of data are collected on the data stack first and then stored into two arrays 1TEST and 2TEST. Data in the arrays are then analyzed to determine the phase difference. A variable DELAY controls the rate of sampling.

This design was implemented at the beginning of the project. At that time, I didn't want to use the B port in NC4000 for the A/D converters, because I thought that the B port might be required to service other devices. As the project progressed, it was clear that B port would not be used. The Datel A/D converters were borrowed from another project for evaluation. Later we got our own National ADC0820 converters, and I decided to use the B port to control the converters directly. The final design of the A/D system, with a quad Op-Amp LM324 conditioning the input signals, is shown in Figure 6.15.

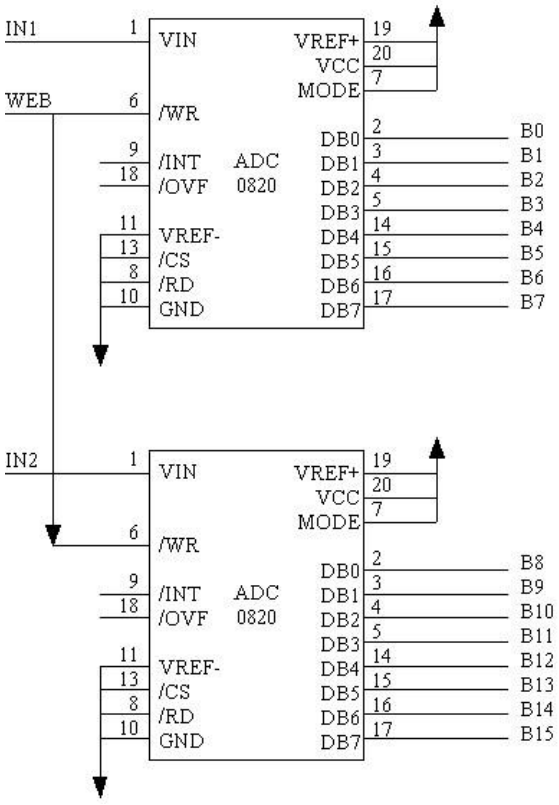


Figure 6.15. A/D Conversion with National ADC0820.

ADC0820 converter is an 8 bit converter also. One is connected to BO-B7 and the other to B8-B15. The B port write-enable line WEB is used to strobe both converters. This configuration

eliminates all the '244's and we have a much simpler and faster system.

The code to operate this A/D converter system is shown in the second screen in Figure 6.16. The data from both converters are grabbed and stored directly into memory. The advantage is that more data can be acquired for analysis if necessary and it is not limited by the depth of the data stack. Since the data from two converters are packed into 16 bit memory cells, they have to be separated and put into two storage arrays.

The A/D converters used here are not the best devices to show off NC4000. Their conversion rate is about 1 MHz, and NC4000 has to wait for the conversion to be completed before it can read the data. Chuck Moore told me he used a video flash converter which is faster than NC4000 so that data is available whenever NC4000 is ready. Nevertheless, both designs discussed here are adequate for analyzing audio signals up to 20 kHz as required by the application.

```
 ( DATA ACQUISITION, 04JUN86CHT ) HEX
A/D   Read 120 pairs of data fros two AID converters and put
      the date on the data stack.
DIGEST  Retrieve AID data on stack and put the* back into the
      1TEST and 2TEST arrays.

ACQUISITION  Acquire data and process them to obtain 20 bucket
      values in RESULTS array.

 ( DATA ACQUISITION, 04JUN86CHT ) HEX
: A/D   78 FOR   -1 9041 !   DELAY 3 FOR NEXT
   8000 @   8001 @   NEXT ;
: DIGEST   78 FOR   FF AND 2TEST I + !
   FF AND 1TEST I + !   NEXT ; DECIMAL
: ACQUISITION   A/D DIGEST 20-PASSES ;

 ( DATA ACQUISITION, IOAUG86CHT
HEX
: A/D ( 6E7 SAMPLES INTO 1TEST ARRAY )
   0 1iEST 1 - ( DATA ADDR )   SAMPLES @
   FOR   0 8 I!   1 ( DELAY ) @ FOR NEXT   1 !+   8 I@
   SWAP   NEXT   2DROP ;
: DIGEST   SAMPLES @ 1 - FOR
   1TEST I + DUP @ DUP >R   FF AND SWAP !
   R> 6 TIMES [ 8001 , ( 21 ) ]   FF AND 2TEST I + !   NEXT ;
DECIMAL
: ACQUISITION   A/D DIGEST 20-PASSES ;
```

Figure 6.16. NC4000 Code for A/D Conversion.

6.14. Fast Byte Flip

NC4000 is a word machine. It takes only two cycles to fetch a 16 bit word from memory, but it takes 10 to 20 cycles to get a byte from memory, as shown in the code of C@. This is clearly a problem if you wanted to use NC4000 to store and process large amount of data in bytes. How would you like a way to swap the two bytes in a word real fast?

There is no free lunch. What you have to give up in this case is the B-port in NC4000 to implement this fast byte swapping engine. B-port must be hardwired, connecting the upper byte with the lower byte; i.e., BO-BS, B1-B9, ... , and B7-B15. When the upper byte in B-port is set for outputing and the lower byte for inputing, the upper byte is flipped to the lower byte. If the I/O assignment is reversed, the lower byte will be flipped to the upper byte.

The code to flip bytes is shown in Figure 6.17.

The words HI>LO and LO>HI configure the data register, mask register and the direction register in the B-port so that data can be sent out through half of the B-port and read back by the other half. After either is executed to set up the B-port, FLIP or FF can be executed to flip the bytes in the T register. FLIP first write the content of T to the B-port and then read the B-port back into T. It takes two cycles to flip T. The word FF does this flipping in one cycle. It uses the special register exchange instruction in NC4000, I@!, which exchanges the T register with the data register in B-port. Because the T to B-port action occurred before the B-port to T transfer, what you get back in the T register is a byte flipped to the other side of the 16 bit word.

```
DECIMAL
: OUTPUT ( n --, setup B-port to output n)
   0 9 I! -1 10 I! 0 11 I! 8 I! ;

: HI>LO ( configure B-port to copy upper byte to lower)
   OUTPUT -256 10 I! 255 9 I! ;

: LO>HI ( configure B-port to copy lower byte to upper)
   OUTPUT 255 10 I! -256 9 I! ;

: FLIP ( n -- n', flip a byte)
   8 I!   8 I@ ;

OCTAL
: FF ( n -- n' one cycle flip )
   [ 157710 ] ; ( 8 I@! )
DECIMAL
```

Figure 6.17. Byte Flipping.

6.15. More Vocabularies

In simplifying cmForth, Chuck Moore threw away the 8 way hashed vocabulary structure he developed for po1yForth. He retained only two vocabularies: FORTH and COMPILER. In doing so, he also discovered that it was not necessary to make the compiler directives, such as IF, ELSE, THEN, BEGIN, UNTIL, etc., immediate words. The immediate words must be executed even during compilation. Since the new compiler in cmForth first searches the COMPILER vocabulary and executes any word it finds in this vocabulary, immediacy is implied and needs not to be declared explicitly. Eliminating the entire class of immediate words and the concept of immediacy is one of the unique features in cmForth.

However, you are left with only two vocabularies, and the COMPILER vocabulary behaves quite differently from the FORTH vocabulary because the words in COMPILER vocabulary cannot be compiled naturally. It would be very nice if youcan build many more vocabularies for large applications.

Let us first take a look at the bottom of the RAM memory and see how these memory words are allocated in Figure 6.18, which is Screen 12 in the source code of cmForth. Chuck kindly left the first 16 words free for us users. The system variables start at location 10H from PREV, OLDEST and so on. He exhausted the 32 local memory words at C/B. The next two words at 20H were reserved for the interrupt vector. The following two words are used by the COMPILER and FORTH vocabularies to store the link pointers pointing to the last words defined in these vocabularies. This is the vocabulary link table. The last system variable is CONTEXT, which contains a 1 if you are searching FORTH vocabulary, or a 2 if you are searching the COMPILER vocabulary. If you reserve more cells below CONTEXT, they can be used to build new vocabularies.

One problem is that INTERPRET and the compiler] knows only two vocabularies, FORTH and COMPILER, and it uses their hash code 1 and 2 explicitly in doing the dictionary search. We have to replace the phrase 1 -FIND in] by the phrase CONTEXT @ -FIND and the phrase 1-' in INTERPRET by the phrase CONTEXT @ -' so that the context vocabulary can be searched. You have to recompile cmForth with these code modifications and the extension of the vocabulary link table.

The space below CONTEXT can be used to construct many vocabularies by storing vocabulary link pointers. If we reserve 10 cells for this table, we will be able to declare 8 vocabularies in addition to FORTH and COMPILER. To build the fifth new vocabulary, for example, we have to define the vocabulary similar to the definitions of FORTH and COMPILER:

```
: APPLICATION  5 CONTEXT ! ;
```

and initialize the vocabulary pointer so that the new vocabulary will be appended to the current FORTH vocabulary:

```
CONTEXT 1 - @ ( pointer to top of FORTH vocabulary)
```

CONTEXT 5 - ! (make APPLICATION a branch on FORTH trunk)

After this, executing APPLICATION will cause this vocabulary to be searched before FORTH.

Rick Van Norman observed that the vocabulary link table does not have to be below CONTEXT. In fact it can be defined as an array anywhere in the RAM memory. To switch context, you have to store the offset of an entry in this table from CONTEXT into CONTEXT. Consequently, you do not have to change the vocabulary structure in cmForth. Only INTERPRET and] must be modified as discussed above. Vocabulary link table can be built whenever it is needed at run time.

```
  ( RAM allocation)  OCTAL
{ : ARRAY ( n)  CONSTANT 154462 USE ;
HEX 10 CONSTANT PREV     ( Last referenced buffer)
    11 CONSTANT OLDEST   ( Oldest loaded buffer)
    12 ARRAY BUFFERS     ( Block in each buffer) I
2 1 - CONSTANT NB       ( Number of buffers)  T

{ 14 CONSTANT CYLINDER  } 15 CONSTANT TIB

( Initialized)
16 CONSTANT SPAN     17 CONSTANT >IN    { l8 CONSTANT BLK )
19 CONSTANT dA
1A CONSTANT ?CODE    lB CONSTANT CURSOR
{ 1C CONSTANT S0 }  10 CONSTANT BASE      IF CONSTANT H
1F CONSTANT C/B     24 CONSTANT CONTEXT
```

Figure 6.18. RAM Memory Allocation in cmForth

Appendix A. cmForth Sorce Listing

cmFoRTH is the version of Forth I (Chuck Moore) wrote and use. Novix supported its development, and we have placed it in the public domain to provide a good model for NC4016 Forths. It meets my goals, though I realize it may not be preferred by everyone. However, since it can recompile itself, I think it's a good starting point for anyone wishing to change it.

Each program block (1-30) has a shadow block of comments that explains what the code does. It does not explain how it is done - read the code to determine that.

cmFORTH does not conform to any standard. I will be noting the differences against Brodie's Starting Forth. The most notable is the absense of DO. LOOP and +LOOP. I presume FORTHkit builders will use FOR and NFXT .

All the multiply code (* , */ , M*) presumes the top argument (multiplicand) is even. This is a hug in the 4016. Try it. There is a software fix you can add, but it is rarely needed. Most multiplies are even. The code for U*+ and M* (block 9) can be changed (at a cost of 6 cycles):

```
: U*+ ( u r u - 1 h) DUP -2 AND 4 I!
   1 AND IF OVER + THEN 14 TIMES *' ; ( 25-26)
: M* ( n n - 1 h) DUP o< IF VNFGATE THFN 0 SWAP
   DUP -2 AND 4 I! 1 AND IF OVER +
   THEN 13 TTMFS *’ *- : ( 31-37 )
```

The procedure for recompi1ing cmFOhTH is :

1. Load compacting compi1er- from block 1.
2. Edit changes in blocks 1-30.
3. Load block 3 to compile.
4. Type GO to test or Burn PROM from 2000, mapping 2000-200F to 1000-100F and 2010-27FF to 0010-07FF

I suggest you compile relocated code as delivered, and compare the code compiled with that in PROM. That is, compare 600 cells from 2025 with 0025. This verifies your source. Then compile changed code and test it. That is, type GO . This is typical of testing changes before burning PROMs.

Looking through the 175 words at the,back of Starting Forth, I note the fol1owing exceptions in cmFORTH:

- Hardware addressing is by cells. Byte addresses are restricted to the first 32K ce11s; even bytes are high.
- Hardware stacks are circular; stark overflow or underflow are neither harmful, detected nor reset. ?STACK ‘S SO are not defined.
- There are exactly two vocabularies, FORTH and COMPILFR. EDITOR . and ASSEMBLER are not defined. COMPIlER words are accessible only in definitions, and are all immediate.
- 1 1+ 1- 2+ 2- are not defined. The compiler optimizes them.
- 2* and 2/ are COMPILER words only.
- type replaces COUNT TYPE and leaves an incremented address.
- PAGE >TYPF -TRAILING 0> C, IMMEDIATE FORGET CMOVE <CMOVF
- CONVERT PAD R# CURRENT are rot defined.
- DOES> is replaced by the phrase DOES R> 7FFF AND

- DUMP takes only an address, displays 8 cells and 1eaaves an incremented address.
- #> acts differently.
- DO LOOP +LOOP /LOOP are replaced by FOR NFXT . 1ndexing is best done with an address on the stack and @+ or !+
- J and I are not defined (stack indexing is expensive).
- LEAVF is replaced by WHILE ... NFXT ... FISF R>… THEN
- [COMPILE] is spelled \ for brevity.
- ?DUP 2DROP 2@ and 2! are the only double or mixed-1ength words.
- LIST COPY WIPE TFXT -TFXT will be defined with an editor.

REFFRENCES

Starting FORTH remains my choice for a Forth text:
Leo Brodie
Startin FORTH
Prentice Hall-1981

C. H. Ting has published an annotated 1isting of cmFORTH:
Footsteps in an Empty Valley. Contact:
Offete Enterprises, 1nc.
156 14th Avenue
San Mateo, CA 94402
(650) 571-7639
He has also formed an NC4000 Users Group and publishes a substantial newsletter: More on the NC4000 Volumes 1, 2, 3 , 4 and 5.

Here is summary of the words defined in cmForth. They are grouped in categories with decreasing frequency of use. This sheet is still being edited for completeness.

Application Words

+ - * /	Binary operators
< > = U<	
AND OR XOR	
M* /MOD	
MOD MIN MAX VNEGATE	
NEGATE ABS 2/MUD	Unary operators
0< 0= NOT	
*/ WITHIN	Trinary operatorS
U*+ M/MOD M/ */MOD	
DUP DROP SWAP OVFR	Stack operators
2DUP 2DROP	
DECIMAL HFX OCTAL	Number base
. .R	Terminal output
EMIT CR SPACF SPACES	
KEY EXPECT	Terminal input
: ; CREATE	Define
VARIABLE CONSTANT	
ALLOT ,	Allot memory
HERE FILL ERASE	

```
@  !  +!                          Memory access
C@  C!  2@  2!
@+  @-  !+  !-
IF  ELSE  THFN  EXIT              Structure
FOR  I  NEXT
BEGIN  WHILE  UNTIL
REPEAT  AGAIN
TIMES  >R  R>

NOP  TWO  CYCIES                  Delay

Interpreter Words
(  Comment
RESET  REMEMBER  EMPTY            Dictionary control
LOAD  THRU                        Interpret
INTERPRET  QUIT
EXECUTE
DOES  USE
LETTER  WORD
-DIGIT  NUMBER
-`  PREVIOUS  USE

PREV  OLDEST  BUFFERS             Variables
BASE  BLK  ?CODE
CNT  >TN  dA  C/R  WIDTH
MSG  CURSOR  H  CONTEXT

NC4000 Words

/'  /''  *'  *-  *F  S'          Op codes
D2*  D2/
I@  I!  I@!                       Internal access
-M/MOD  M*+                       Arithmetic

FormattinG Words
TYPE                              Termina1 output
HOLD  DIGIT  (.)
<#  SIGN  #  #S
U.R  U.  DUMP
ABORT"  ."
RX                                Terminal input

Disk Words
BLOCK  BUFFER  UPDATE
FLUSH  EMPTY-BUFFERS  IDENTIFY

Compiler words
,C  ,A  \\
[  ]  LITERAL
COMPILE  \
SMUDGE  RECURSIVE
-SHORT  FIX  -SHORT

Headless Words
abort"  dot"                      Terminal output
ADDRESS  ABSENT  UPDATED          Buffer management
ESTABLISH   ##  buffer  block
```

```
-LETTER 10*+                    interpreter
SAME  HASH  -FIND
ROM  BAUD  Reset
COUNT                          Compiler
OR,

Dubious Words                  (of doubtful value)
ROT                            Rotate top of stack
C@+                            Unpack 2 characters
MOVE                           Move words
OFFSFT                         Block offset
?    @ .
```

```
1 ( FORTNkit: 1987 December)
2    ( Separated Heads)
3 ( cmFORTH )   EMPTY
4 ( Optimizing compiler)   OCTAL
5    ( Defining Words)   OCTAL
6    ( Binary operators)    OCTAL
7 ( Nucleus)   OCTAL
8
9 ( Multiply, divide)
10    ( Memory reference operators)
11    ( Words)
12    ( Ram allocation)   OCTAL
13 ( ASCII terminal: 4X in. OX out) HEX
14    ( Serial EXPECT)   HEX
15    ( Numbers)
16    ( Strings )  HEX
17    ( 15-bit buffer manager)
19    ( Disk rond/wrlte)
19 ( Interpreter)
20    ( Dictionary search)
21    ( Number ir?,ut) HEX
22    ( Control)
23    ( Initializo) HEX
24    ( Word s )
25 ( Compiler)   OCTAL
26    ( Compiler)   OCTAL
27    ( Defining words)   OCTAL
23    ( uCODF )  OCTAL
29    ( Structures)   OCTAL
30    ( Strings)   HEX
```

1

```
cmFORTH shadow blocks (1987 December). Addresses are hex:
    word timing in parentheses after ; ( cycles)
.
3 LOAD  compiles the compacting compiler (blocks 4-6). Block 6
exits in COMPILFR vocabulary, anticipating additions.
0<  is redefined to resolve timing conflict.
END  terminates a definition.
REMEMBER;  saves vocabulary heads (at compile time).
FORTH  puts following words in interpretive vocabulary.
-MOD  provides modular arithmetic.  It does a subtract if the
    result is non-negative.

THRU  loads a sequence of blocks.
EMPTY  empties the dictionary except for compacting compiler.
```

2

```
H'  holds the next available address in the target dictionary.
2000  relocates target addresses from RAM (2000) to PROM (0).
{   }  switches between host and target dictionary by exchanging
pointers and relocation offsets.
COMPILER }  compiles an indirect reference for a headless word.
FORGET  smudges a word that cannot execute in target dictionary.
RECOVER  recovers a return (after AGAIN )

SCAN  finds the next word in target dictionary.
TRIM  relocates the vocabulary link and erases the smudge bit.
CLIP  constructs a target vocabulary and stores its head.

PRONE  relinks the target dictionary to produce a stand-alone
application (fixing the end-of-vocabulary word)
and restores the host dictionary.
```

3

```
3 LOAD  recompiles cmFORTH. EMPTY clears dictionary for a new
    application.
2 LOAD  compiles the target compiler.
Target is compiled at 2000 which is initialized to 0.
BOOT copies PROM to RAM at power-up. The reference to -1
    disables PROM and enables RAM (setting A15 clocks 74).
Low RAM (16-24) is initialized (see block 12).

#  is the bottom of the target dictionary. PRUNE changes its
name to null and link to 0. This version of EXIT marks the
end of both vocabulary chains.
The address of RESET is relocated into the end of BOOT .
The end of target program is stored into TIB and HERE  .
COMPILER head is selected for PRUNE
GO  emulates BOOT for testing: 3 LOAD GO
```

1

```
 ( FORTHkit 1987 December)


( Optimizing compiler)  4 LOAD 5 LOAD 6 LOAD


: 0<  \ 0<  \ NOP ;
: END  \ RECURSIVE  POP DROP ;
: REMEMBER;  CONTEXT 2 - 2@  \ END ;
   FORTH
: -MOD ( n n - n)  4 1!  MOD' ; ( 3)

: THRU ( n n 1  OVER -  FOR  DUP LOAD 1 +  NEXT DROP ;
: EMPTY  FORGET REMEMBER;
```

2

```
 ( Separated heads)
      VARIABLE H' HEX 2000 ,( relocation)
      : {  dA @  HERE  H' 2@ H !  dA !  H' 2! ;  : }  { ;
COMPILER : }  H' @.A \\  PREVIOUS  8000 XOR SWAP !  { ;
FORTH : forget  SMUDGE ;
      : RECOVER  -1 ALLOT ;

: SCAN ( a - a)  @ BEGIN  DUP 1 2000 WITHIN  WHILE @  REPEAT ;
: TRIM ( a a - a)  DUP PUSH  dA @ - SWAP !  POP
   DUP 1+  DUP @ DFFF AND  OVER !
   DUP @ 200 /  F AND +  DUP @ FF7F AND  SWAP ! ;
: CLIP ( a)  DUP BEGIN  DUP SCAN DUP  WHILE  TRIM  REPEAT
   2025 XOR  dA @ -  SWAP ! @ , ;
: PRUNE  {  CONTEXT 2 -  DUP CLIP  1 + CLIP }
   20 0 2025 2! EMPTY ;
```

3

```
 ( cmFORTH)  EMPTY
( Target compiler) 2 LOAD
HEX 2000 800 0 FILL  2000 H' !
: BOOT }  16 FFF FOR  0 @+  1 !+  NEXT  -1 @ ( reset) ;
   0. 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 ,  0 ,  0( TIB) ,
   77C0 , 0 , 0 , 0 , 0 , 0 ,  1FF ( SO) ,  A ( BASE) ,
   0 ( H) ,  DECIMAL 521 ( C/B 5MHz 9600b/s) ,
   { : interrupt }  POP DROP ;  0 , 0 , 1( CONTEXT) ,

( Nucleus)  : #  POP DROP ;  7 11 THRU
( Interpreter) 12 22 THRU
( Initialize) 23 24 THRU  ` reset dA @ -  HEX 2009 !  DECIMAL
( Compiler) 25 30 THRU  } PRUNE
: GO  FLUSH  [ HEX ]  2015 4 I!  15 6EA FOR
   4 I@!  1 @+ 4 I@!  1 !+  NEXT  2009 PUSH ;
```

4

```
FORTH  sets interpretive vocabulary for both searches and
  definitions.  Words are compiled in definitions.
COMPILER  sets immediate vocabulary. Words are executed in :
uCODE  names a NC4016 micro-coded instruction. Compiled on use.
\  compiles a following compiler directive (that would normally
  be executed).  Named  [COMPILE]  in FORTH-83.
4016 instructions:
  !-  stores and decrements.  I@! exchanges stack&register.
  NOP delays 1 cycle.         TWO delays 2 cycles.
  O+c Adds 0 with carry.      MOD' conditionally subtracts R4.
  N!  stores and saves data.  -1 fetches register 3

DUP?  compacts preceding DUP with current instruction. Used
to redefine  I!  and  PUSH  (previously >R ).
```

5

```
PACK  sets the return bit, if an instruction does not reference
  the Return stack.  Otherwise it compiles a return.  It exits
  from  EXIT  with  POP DROP .
EXIT  optimizes return if permitted (?CODF nonzero):
  For instructions (bit-15 = 1) it calls PACK except for jump
    or 2-cycle instructions;
  for calls to the same 4K page, it substitutes a jump.

;  is redefined to use the new EXIT .

CONSTANT  is redefined to take advantage of the new EXIT for
   5-bit literals.
```

6

```
BINARY  defines and compacts ALU instructions. If the previous
  instruction was a fetch (ALU code 7) and not a store or DROP
  the ALU code is merged; stack push is inhibited. Otherwise
  a new instruction is compiled.  ?CODE holds address of
  candidate for compaction.

SHIFT  defines and compacts shift instructions. Shift left
  ( 2* ) and right ( 2/) may be merged with an arithmetic
  instruction;  sign propagate ( 0< } only with DUP .
;
DROP  OR  XOR  AND  +  -  SWAP-  are redefined.
2*   2/  0<  likewise.
```

4

```
( Optimizing compiler)  OCTAL
: FORTH  1 CONTEXT ! ;
: COMPILER  2 CONTEXT ! :
: uCODE ( n)  CREATE ,  DOES  R> 77777 AND  @  ,C ;
COMPILER : \  2 -` IF DROP ABORT" ?" THEN ,A ;
        : !-   172700 SHORT ;
        : I@!  157700 SHORT :
        100000 uCODE NOP      140000 uCODE TWO
        154600 uCODE O+c      102404 uCODE MOD'
        177300 uCODE N!       147303 uCODE -1
  FORTH : DUP?  HERE 2 - @ 100120 = IF
           HERE 1 - @ 7100 XOR -2 ALLOT ,C THEN ;
COMPILER : I!  157200 SHORT DUP? ;
         : PUSH  157201 C DUP? ;
```

5

```
( Defining Words)  OCTAL
FORTH : PACK ( a n- a)  160257 AND  140201  XOR IF
         40 SWAP +!  ELSE DROP 100040 ,  THEN  POP DROP ;
COMPILER : EXIT  ?CODE @ DUP  IF  \\ DUP @ DUP 0<  IF
         DUP 170000 AND 100000 =  IF PACK THEN
         DUP 170300 AND 140300 =  IF PACK THEN
         DUP 170000 AND 150000 =  IF
           DUP 170600 AND 150000 XOR  IF PACK  THEN  THEN DROP
       ELSE DUP HERE  dA @ - XOR  170000 AND 0=  IF
         7777 AND 130000 XOR  SWAP !  EXIT  THEN DROP  THEN
    THEN DROP 100040 , ;
         : ;  \ RECURSIVE  POP DROP  \ EXIT ;
FORTH : CONSTANT ( n)  CREATE -1 ALLOT  \ LITERAL  \ EXIT ;
```

6

```
( Binary operators)  OCTAL
: BINARY ( n n)  CREATE , ,  DOES  POP 77777 AND 2@
   ?CODE @ DUP  IF  @ DUP 117100 AND 107100 =
     OVER 177700 AND  157500 = OR  IF ( y -!)
       DUP 107020 - IF SWAP DROP XOR DUP 700 AND 200 = IF
          500 XOR  ELSE DUP 70000 AND 0= IF  20 XOR  THEN THEN
          ?CODE @ !  EXIT THEN
   THEN THEN DROP  ,C  DROP ;
: SHIFT ( n n)  CREATE , , DOES  POP 77777 AND  2@
  ?CODE @ ?DUP IF  @ DUP 100000 AND = WHILE  ?CODE @ +! EXIT THEN
   DROP THEN  100000 XOR ,C ;
COMPILER 7100 107020 BINARY DROP
         4100 103020 BINARY OR     2100 105020 BINARY XOR
         6100 101020 BINARY AND    3100 104020 BINARY +
         5100 106020 BINARY -      1100 102020 BINARY SWAP-
2 171003 SHIFT 2*   1 171003 SHIFT 2/   3 177003 SHIFT 0<
```

7

```
ROT  is a slow way to reference into the stack.

0=  returns false (0) if stack non-zero; otherwise true (-1).
NOT  same as 0=.  FORTH-83 wants one's complement.
<  > subtract and test sign bit. Range of difference limited
   to 15 bits (-20000 is not less-than 20000).
=  equality tested by XOR.
U<  unsigned compare with 16-bit range (0 is less-than 40000).

{... }  surround words defined into host dictionary. Used
  during compilation, they will not be in target dictionary.
4016 instructions:
   *" multiply step          *- signed multiply step
   D2* 32-bit left shift  D2/ 32-bit right shift
   /' divide step            /'' final divide step
   F* fraction multiply     S' square-root step
```

8

9

```
M/MOD  30-bit dividend; 15-bit divisor, quotient, remainder.
M/  signed dividend; 15-bit divisor, quotient.
VNEGATE  negates both multiplier and multiplicand.
M*  32-bit signed product; multiplier (on top) must be even.


/MOD  15-bit dividend, divisor, quotient, remainder.
MOD  15-bit dividend, divisor, remainder.

U*+  15-bit multiplier, multiplicand, addend: 30-bit product.
*/  signed multiplier, multiplicand, result: 15-bit divisor;
  multiplier (in middle) must be even.
*  signed product; multiplier (on top) must be even.
/  signed dividend, quotient: 15-bit divisor.
```

7

```
( Nucleus)  OCTAL
: ROT ( n n n - n n n)  PUSH SWAP  POP SWAP ; ( 5)

: 0= ( n - t)  IF  0 EXIT  THEN -1 ; ( 3)
: NOT ( n - t)  0= ; ( 4)
: < ( n n – t)  - 0< ; ( 3)
: > ( n n - t)  SWAP- 0< ; ( 3)
: = ( n n – t)   XOR 0= ; ( 5)
: U< ( u u – t)  - 2/ 0< ; ( 3)

{ COMPILER
104411 uCODE *’     102411 uCODE *-
100012 uCODE D2*    100011 uCODE D2/
102416 uCODE /’     102414 Ucode /’’
( 102412 uCODE *F   102616 uCODE S')  FORTH }
```

8

9

```
( Multiply, divide)
: M/MOD ( 1 h u - g r)  4 I!  D2* 13 TIMES  /’  /’’ ; ( 21)
: M/ ( 1 h u - q r)  OVER 0< IF  OUP PUSH +  POP THEN
   M/MOD DROP ; ( 27-30)
: VNEGATE ( v - v)  NEGATE SWAP  NEGATE SWAP ; ( 5)
: M* ( n n - d)  DUP 0< IF  VNEGATE  THEN 0 SWAP
   4 I!  13 TIMES  *’  *- ; ( 26-31)

: /MOD 1 u u - r q)  0 SWAP  M/MOD SWAP ;( 25)
: MOD ( u u – r)  /MOD DROP ; ( 27)

: U*+ ( u r u - l h)  4 I!  14 TIMES  *’ ; ( 20)
: */ ( n n u - n)  PUSH M*  POP M/ : ( 64)
: * ( n n - n)  0 SWAP U*+ DROP ; ( 241
: / ( n u - q)  PUSH  DUP 0<  POP M/ ; ( 35)
```

10

```
2/MOD  16-bit unsigned dividend; 15-bit guotient, remainder.
  \\ (break compaction) used to combine + 2/ ;
+!  adds to memory.
Byte address is 2* cell address; high byte is byte 0. Range
  restricted to low RAM (0-7FFF).
C!  stores 8-bit data into byte address: other byte naffected.
C@  fetches 8-bits from byte address.

2@  fetches 2 16-bit numbers; lower address on top.
2!  stores 2 16-bit numbers.

2DROP  DROP  DROP  ;  is faster and usually no longer.
MOVE  the fastest move that does not stream to-from stack.
FILL fills RAM with constant.
```

11

```
EXECUTE  executes code at an address by returning to it. CYCLES  delays n+4
cycles - count 'em.

2DUP  copies 32-bit (2 16-bit) stack item.
?DUP  copies stack if non-zero.

WITHIN  returns true if number within low (inclusive) and high
   (non-inclusive) limits;  all numbers 16 bits or signed.
ABS  returns positive number (15-bits).
`
MAX  returns larqer of pair; 15-bit range.
MIN  returns smaller.  Intertwining code saves 2 cells; left in
   as illustration of obscure but efficient code.
```

12

```
ARRAY  defines an array that adds an index from stack in only
   2 cycles. Similar to VARIABLE

These low-RAM variables are used by cmFORTH (0-F are unused).
   Change them cautiously! In particular, make sure a variable
   is not used during compilation. For example, HEX is
   redefined to set BASE . It can be used if BASE has not
   moved; otherwise it must be FORGETted.
Non-standard variables:
   ?CODE  address of last instruction compiled. Zero indicates
      no compaction permitted (ip, after THEN ).
   dA  offset to be added to compiled addresses. Normally 0.
      Relocated code cannot be executed!
   CURSOR  tracks cursor (terminal dependent); used by EXPECT
   SO  serial output polarity: 1FF or 200.
   C/B  cycles/bit for serial I/0.
```

10

```
  ( Memory reference operators)
: 2/MOD ( n - r q)  DUP 1 AND  SWAP 0 [ \\ ] + 2/ ; ( 6)

: +! ( n a)  0 @+ PUSH  +  POP ! ; ( 9)
: C! ( n b)  2/MOD  DUP PUSH @  SWAP IF  -256 AND
   ELSE  255 AND  SWAP 6 TIMES 2*  THEN XOR  POP ! ; ( 20-29)
: C@ ( b - n)  2/MOD @  SWAP 1 - IF  6 TIMES 2/  THEN 255 AND ;
  ( 10-20)
: 2@  ( a - d)  1 @+  @ SWAP ; ( 6)
: 2! ( d u)  1 !+  ! ; ( 6)
{ OCTAL COMPILER : -ZERO  1 +  \ BEGIN  130000 , ;  FORTH }
: MOVE ( s d n)  PUSH  4 I!  BEGIN -ZFRO
      1 @+  4 I@!  1 !+  4 I@!  THEN NEXT DROP ; ( 7* 5+)
: FILL ( a n n)  4 I!  FOR -ZERO 4 I@ SWAP  1 !+  THEN NEXT
   DROP ; ( 5* 8+)
```

11

```
  ( Words)
: EXECUTE ( a)  PUSH ; ( 3)
: CYCLES ( n)  TIMES ; ( 4 n+)

: ?DUP ( n - n n. 0)  OUP IF  DUP EXIT  THFN ; ( 4)
: 2DUP ( d - d d)  OVER OVER ; ( 3)
: 2DROP ( d)  DROP DROP ; ( 3)

: WITHIN ( n l h - t)  OVER - PUSH  - POP U< ;
: ABS ( n - u)  DUP 0<  IF  NEGATE EXIT  THEN ; ( 4)

: MAX ( n n - n)  OVER OVER - O< IF  BEGIN  SWAP DROP ;
: MIN ( n n - n)  OVER OVER - O< UNTIL  THEN DROP  ; ( 6)
```

12

```
  ( RAM allocation)  OCTAL
{ : ARRAY ( n)  CONSTANT  154462 USE ;
HEX 10 CONSTANT PREV     ( Last referenced buffer)
    11 CONSTANT OLDEST   ( Oldest loaded buffer)
    12 ARRAY BUFFERS     ( Block in each buffer) I
N2 1 - CONSTANT NB       ( Number of buffers)  T

{ 14 CONSTANT CYLINDER  } 15 CONSTANT TTB

( Initialized)
16 CONSTANT SPAN     17 CONSTANT >IN     { 18 CONSTANT BLK }
19 CONSTANT dA
1A CONSTANT ?CODE    1B CONSTANT CURSOR
{ 1C CONSTANT S0 }   1D CONSTANT BASE      1E CONSTANT H
1F CONSTANT C/B      24 CONSTANT CONTEXT
```

13

EMIT sets Xmask to lE so that only XO can be changed. Start/
   stop bits are added and polarity set. I! emits bits at C/B
   rate thru X0.
CR emits carriage-return and line-feed.

TYPF types a string with prefixed count byte. It returns an
   incremented cell address. This is not FORTH-83 standard.

RX reads a bit from pin X4.
KEY reads 8-bits from X4. It waits for a start bit, then
   delays until the middle of the first data bit. Each bit is
   sampled then ored into bit 7 of the accumulated byte. It
   does not exit until the stop bit (low) is detected.

14

SPACE emits a space.
SPACES emits n>0 spaces.
HOLD holds characters on the stack, maintaining a count.
   It reverses the digits resulting from number conversion.

EXPECT accepts keystrokes and buffers them (at TIB). An 8
   will discard a character and emit a backspace: a D will
   emit a space and exit; all other keys are stored and echoed
   until the count is exhausted. Actual count is in SPAN .

15

DIGIT converts a digit (0-F) into an ASCII character.
<# starts conversion by tucking a count under the number.
#> ends conversion by emitting the string of digits.
SIGN stacks a minus sign, if needed.
converts the low-order digit of a 16-bit number.
#S converts non-zero digits, at least one.
(.) formats a signed number.
. displays a 16-bit signed integer, followed by a space.
U.R displays a right-justified 16-bit unsigned number.
U. displays an unsigned number.
DUMP displays an address and 3 numbers from memory. It
   returns an incremented address for a subsequent DUMP .

13

```
( ASCII terminal: 4X in, OX out) HEX
: EMIT ( n)  lE D I!  2* SO @ XOR
   9 FOR  DUP C I!  2/  C/B @ A - CYCLES  NEXT DROP ;
: CR  D EMIT  A EMIT ;
: TYPE ( a - a)  2* DUP C@ 1 - FOR  1 +  DUP C@ EMIT  NEXT
   2 + 2/ ;

{ : RX ( - n) }  C I@  10 AND ; ( 3)
: KEY (- n)  0 BEGIN RX  10 XOR UNTIL  C/B @  DUP 2/ +
   7 FOR  10 - CYCLES  2/  RX  2* 2* 2* OR  C/B @  NEXT
   BEGIN RX UNTIL  DROP ;
```

14

```
  ( Serial EXPECT)  HEX
: SPACE  20 EMIT ;
: SPACES ( n)  0 MAX  FOR -ZER0  SPACE THEN NEXT ;
: HOLD ( ..# x n - ..# x)  SWAP PUSH  SWAP 1 + POP ;

: EXPECT ( a # )  SWAP CURSOR !
   1 - DUP FOR  KEY DUP 8 XOR IF
          DUP D XOR IF  DUP  CURSOR @ 1 !+  CURSOR !  EMIT
          ELSE  SPACE  DROP  POP - SPAN ! EXIT THFN
      ELSE ( 8)  DROP  DUP I XOR  [ OVER ]  UNTIL
           CURSOR @ 1 -  CURSOR !  POP 2 + PUSH  8 EMIT
      THEN NEXT 1 + SPAN ! ;
```

15

```
  ( Numbers)
: DIGIT ( n- n)  DUP 9 >  7 AND +  48 + ;
: <# ( n - ..# n)  -1 SWAP ;
: #> ( ..# n)  DROP FOR EMIT NFXT ;
: SIGN ( ..# n n -..# n)  0< IF  45 HOLD THEN ;
: # (..# n -..# n) BASE @ /MOD  SWAP DIGIT HOLD ;
: #S ( ..# n - ..# 0)  BEGIN  #  DUP O= UNTIL ;
: (.) ( n - ..# n)  DUP PUSH ABS  <# #S  POP SIGN ;
: . ( n)  (.) #> SPACE ;

: U.R ( u n)  POSH  <# #S  OVER POP SWAP-  1 - SPACES  #> ;
: U. ( u)  0 U.R  SPACE ;
: DUMP ( a - a)  CR  DUP 5 U.R SPACE  7 FOR
   1 @+ SWAP  7 U.R  NEXT SPACE ;
```

16

```
i v HERE returns next address in dictionary.

abort"  types the current word (at HERE ) and an error message
   (at  I )  It also returns the current BLK to locate an
   error during LOAD .  It will end with QUIT , when defined.
   It is a headless definition. referenced only bv ABORT"
dot"  types a message whose address is pulled off the return
   stack, incremented and replaced

ABORT"  compiles abort" and the following string. This is a
   host COMPILER definition. The target definition is in
   block 30.
."  compiles dot" and the following string.
```

17

```
PUFFERS  returns indexed address of buffer ID. PRFV  current buffer number
(0-NB).
OLDEST  last buffer read.  Next buffer i s OLDEST @ 1 + .
ADDRESS  calculates a buffer address from buffer number. NB is
   1.  If increased. ADDRFSS and BUFFERS must be also.
ABSENT  returns the block number when the requested block isn' t
   already in RAM. Otherwise it returns the buffer address and
   exits from BLOCK .
UPDATED  returns the buffer address and current block number if
   the pending buffer has been UPDATEd . Otherwise it returns
   the buffer address and exits from the calling routine
   ( BLOCK or BUFFER ).  Pending means oldest but not Just used.

UPDATE  marks the current buffer ( PRFV ) to be rewritten. ESTABLISH  stores
the block number of the current buffer.  _
IDENTIFY  stores a block number into the current buffer.
   Used to copy blocks.
```

18

```
##  emits 3 bytes to host to start a block transfer:  0 followed
   by block number.

Buffer  transmits an updated block and awaits acknowledgement.
BUFFER  returns address of an empty (but assigned) buffer.
:
block  reads a block.
BLOCK  returns the buffer address of a specified block, writing and
   reading as necessary.

FLUSH  forces buffers to be written.
EMPTY-BUFFERS  clears buffer ID’s, without writing.
```

16

```
  ( Strings )  HEX
: HERE (-a)  H @ ;
{ : abort" }  H @ TYPE  SPACE  POP 7FFF AND TYPE  2DROP
   BLK @  ?DUP DROP  0 ( QUIT) ;
{ : dot" }  POP 7FFF AND  TYPE  PUSH ;

{ COMPILER : ABORT"  COMPILE abort"  22 STRING ;
           : ."  COMPILE dot"  22 STRING ;
      FORTH }
```

17

```
  ( 15-bit buffer manager)
{ : ADDRESS ( n – a) }  30 +  8 TIMES 2* ;
{ : ABSENT ( n – n) }  NB FOR  DUP  I BUFFERS @ X0R  2* WHILE
     NEXT EXIT THEN POP  PREV N!  POP DROP SWAP DROP  ADDRESS ;
..
{ : UPDATED ( - a n ) }  OLDEST @  BEGIN  l +  NB AND
       DUP PKEV @ XOR UNTIL  OLDEST N!  PRFV N!
    DUP ADDRESS  SWAP BUFFERS  DUP @
    8192 ROT !  DUP 0< NOT IF  POP DROP DROP  THEN ;

: UPDATE  PREV @ BUFFERS  0 @+ SWAP  32768 OR  SWAP ! ;
{ : ESTABLISH ( n a – a) }  SWAP  OLDEST @  PREV N!
   BUFFERS ! :
: IDENTIFY ( n a - a)  SWAP  PREV @ BUFFERS
```

18

```
  ( Disk read/write )  ^~
{ : ## ( a n - a a #1 }  0 EMIT  256 /MOD EMIT EMIT  DUP 1023 ;

{ : buffer ( n - a) }  UPDATED
   ## FOR  1 @+ SWAP  EMIT  NEXT  KEY 2DROP ;
: BUFFER ( n - a)  buffer ESTABLISH ;

{ : block ( n a - n a) }  OVER  ## FOR  KEY  SWAP 1 !+
     NEXT DROP ;
: BLOCK ( n – a)  ABSENT buffer  block ESTABLISH ;

: FLUSH  NB FOR  8192 BUFFER DROP  NEXT ;
: EMPTY-BUFFERS  PREV [ NB 3 + ] LITERAL  0 FILL  FLUSH ;
```

19

```
LETTER  moves a string of characters from cell address a to
   byte address b .  Terminated by count ( # ) or delimiter  '
   (register 6).  Input pointer >IN is advanced.

-LFTTER  scans the source string for a non-delimiter.  If found,
   calls LETTER .

WORD  locates text in either block buffer or TIB  ( BLK  is 0)
   Reads word into HERE prefixing count and suffixing a space
   (in case count even).
```

20

```
SAME  compares the string at HERE with a name field. Cell
   count is in register 6.  High bit of each cell is ignored.
   Returns address of parameter field: requires indirect
   reference if high bit of count set (separated head).

COUNT  extracts the cell count from the first word of a string.
HASH  returns the address of the head of a vocabulary.
-FIND  searches a vocabulary for match with HERE . Fails with
   zero link field.
```

21

```
-DIGIT  converts an ASCII character to a digit ( 0-Z).
   Failure generates an error message.

C@+  increments address in register 6 and fetches character.
10*+  multiplies number by BASE and adds digit.
NUMBER  converts given string to binary:  stores BASE in R4:
   saves minus sign:  terminates on count;  applies sign
```

19

```
( Interpreter 1
{ : LETTER ( b a # - b a) }  FOR DUP @  6 I@ XOR  WHILE
      1 @+ PUSH OVFR C!  1 + POP NEXT  EXIT THEN
      >IN @ POP -  >IN ! ;
{ : -LETTER ( b a # - b a) } ?DUP IF
      1 - FOR  1 @+ SWAP  6 I@ XOR  0= WHILE NEXT  EXIT THFN
      1 - POP LFTTFR  THEN ;
: WORD ( n - a)  PUSH H @ DUP 2* DUP 1 +  DUP >IN @
      BLK @  IF BLK @ BLOCK +  1024  ELSE TIB @ + SPAN @  THEN
      >IN @ OVER >IN !  - POP 6 I!
      -LETTER DROP  32 OVER C!  SWAP- SWAP C! ;
```

20

```
  ( Dictionary search)
{ : SAME ( h a - h a f. a t) }  OVER 4 I!  DUP 1 +
      6 I@  FOR l @+ SWAP  4 I@ 1 @+ 4 I!  - 2*  IF
         POP DROP  0 AND EXIT  THEN
      NEXT SWAP 1 + @  0<  IF  @  THEN SWAP ;

{ : COUNT ( n - n) }  7 TIMES 2/  15 AND ;
{ : HASH ( n - a) }  CONTEXT SWAP- ;
{ : -FIND ( h n - h t. a f) }  HASH OVER @ COUNT  6 I!
      BEGIN  @ DUP WHILE  SAME UNTIL  0 EXIT THEN  -1 XOR ;
```

21

```
  ( Number input)  HEX
: -DIGIT ( n - n)  30 -  DUP 9 > IF  7 -  DUP A < OR  THEN
   DUP BASE @ U< IF  EXIT THEN
   2DROP  ABORT" ?” ; RECOVER

{ : C@+ ( - n) }  6 I@ 1 +  DUP 6 I!  C@ ;
{: 10*+ ( u n - u) } -DIGIT  OE TIMES *’  DROP ;
: NUMBER ( a – n)  BASE @ 4 I!  0 SWAP 2*  DUP 1 + C@ 20 =
   PUSH DUP 1 - 6 I!  C@ I +  1 - FOR  C@+ 10*+  NEXT
   POP  IF  NEGATE  THEN ;
```

22

-` searches vocabulary for following word.
   returns address of following word in current vocabulary
   Error message on failure. FORGET to use host version.

INTERPRET accepts block number and offset. Searches FORTH
   and executes words found; otherwise converts to binary.

QUIT accepts a character string into the text input buffer,
   interprets and replies ok to signify success; repeats.
   The address of QUIT is relocated into the end of abort" .

23

FORGET restores HERE and vocabulary heads to values saved at
   compile time (by REMEMBFR; 1.

BPS awaits a start bit, assumes only the first data bit is
   zero and computes C/B . Type a B or other even letter.
RS232 examines the serial input line and inverts serial I/0 if
   an inverting buffer is used (line rests low).

Reset is executed at power-up or reset.
   Carefully initializes I/0 registers to avoid glitches.
   Empties buffers at power-up only (TIB contains garbage).
   Calibrates serial i10.
   Cheerful hi and awaits command.

24

This is the beginning of the compiler. A turn-key application
   might need only the code above.

Common words are defined for both interpreter and compiler.

Number base words defined together: DECIMAL required.

LOAD saves current input pointers. Calls INTERPRET . restores
   Input pointers and returns to DECIMAL . >IN and BLK are
   Treated as a 32-bit pointer. FORGET so that host LOAD
   Will be used.

22

```
  ( Control)
: -` ( n - h t. a f)  32 WORD  SWAP -FIND ;
: ` ( - a)  CONTEXT @ -` IF  DROP ABORT" ?"  THEN ;  forget

: INTERPRET ( n n)  >IN 2!  BEGIN  1 -` IF  NUMBER
   ELSE  EXECUTE  THEN AGAIN ;  RECOVER
: QUIT  BEGIN CR  TIB @ 64 EXPECT
   0 0 INTERPRET  ." ok"  AGAIN ;  RECOVER

` QUIT  dA @ -  ' abort"  1l + !
```

23

```
  ( Initialize)  HEX
: FORGET ( a)  POP 7FFF AND  DUP 2 + H !  2@ CONTEXT 2 - 2!
   1 CONTEXT ! ;

{ : BPS }  4 BEGIN RX  10 XOR  UNTIL  BEGIN 5 +  RX UNTIL
   2/ C/B ! :
{ : RS232 }  RX IF  EXIT THEN  200 SO !  0B C I! ;

{ : reset }  0 ( RESET)
   0 DUP 9 I!  DUP A I!  DUP 0B I!  DUP 8 I!  -1 A I!
   DUP D I!  DUP E I!  F I!  1A C I!
   TIB 2@ XOR IF  EMPTY-BUFFFRS  SPAN @ TIB !  THEN
   RS232  F E I!  BPS  ." hi"  QUIT ;
```

24

```
  ( Words)
: SWAP  SWAP ;    : OVER  OVER ;
: DUP  DUP ;      : DROP  DROP ;

: XOR  XOR ;      : AND  AND ;
: OR  OR ;
: +  + ;          : -  - ;
: 0<  0< ;        : NEGATE  NEGATE ;

: @  @ ;          : !  ! ;

: OCTAL  8 BASE ! ;  forget
: DECIMAL  10 BASE ! ;  forget
: HEX  16 BASE ! ;  forget
: LOAD ( n)  >N 2@  PUSH PUSH  0 INTERPRET  10 BASE !
    POP POP >N 2! ;  forget
```

25

\\ breaks code compaction.
ALLOT increments the dictionary pointer to allot memory.
, compiles a number into the dictionary.
,C compiles an instruction available for compaction.
,A compiles a address relocated by dA .
LITERAL compiles a number as a short literal, if possible.
[stops compilation by popping the return stack, thus returning
   out of the infinite] loop.

] unlike INTERPRET , searches both vocabularies before falling
   into NUMBER . When a word is found in COMPILER it is
   executed; if found in FORTH it is compiled. If it is a
   single instruction, it is placed in-line; otherwise its
   address is compiled for a call.

26

PREVIOUS returns the address and count of the name field of
   the word just defined.
USE assigns to the previous word a specified code field.
DOES provides a behavior for a newly defined word. It is
   executed when that word is defined.
SMUDGE smudges the name field to avoid recursion.
EXIT returns from a definition early (FORTH version).

COMPILE pops the address of the following word and compiles it.
   7FFF AND masks the carry bit from the return stack.
EXIT compiles a return instruction (COMPILEP version).
RECURSIVE unsmudges the name field so a new word can be found.
: terminates a definition. FORGET permits more definitions.

27

CREATE creates an entry in the dictionarv. It saves space for
   the link field, then fetches a word terminated by space. It
   links the word into the proper vocabulary, allots space for
   the name field and compiles the return-next-address
   instruction appropriate for a variable.

: creates a definition: -1 ALLOT recovers the instruction
   compiled by CREATE :] compiles the definition in its place.
   FORGET permits more definitions.

CONSTANT names a number by compiling a literal.

VARIABLE initializes its variable to zero.

25

```
 ( Compiler)  OCTAL
: \\  0 ?CODE ! ;
: ALLOT ( n)  H +!  \\ ;
: , ( n)  H @ !  1 H +! ;
: ,C ( n)  H @ ?CODE !  , ;
: ,A ( a)  dA @ -  ,C ;
COMPILER : LITERAL ( n)  DUP -40 AND  IF 147500 ,C  ,  EXIT
   THEN  157500 XOR ,C ;
: [  POP DROP ;

FORTH : ]  BEGIN 2 -'  IF  1 -FIND  IF  NUMBER  \ LITERAL
   ELSE DUP @
      DUP 140040 AND  140040 =  OVER 170377 AND  140342 XOR AND
      SWAP 170040 AND  100040 =  OR  IF @ 40 XOR  ,C
      ELSE  ,A THEN  THEN
   ELSE  EXECUTE  THEN AGAIN ;  RECOVER
```

26

```
  ( Compiler)  HEX
: PREVIOUS ( - a n)  CONTEXT @ HASH  @ 1 +  0 @+ SWAP ;
: USE ( a)  PREVIOUS  COUNT +  1 +  ! ;
: DOES  POP 7FFF AND  USE ;
: SMUDGE  PREVIOUS  2000 XOR  SWAP ! ;
: EXIT  POP DROP ;

: COMPILE  POP 7FFF AND  1 @+ PUSH  ,A ;
OCTAL
COMPILER : EXIT  100040 ,C ;   HEX
         : RECURSIVE  PREVIOUS  DFFF AND  SWAP ! ;
         : :  \ RECURSIVE  POP DROP  \ EXIT ;  forget
```

27

```
  ( Defining words)  OCTAL
FORTH : CREATE  H @ 0 ,  40 WORD  CONTEXT @ HASH
   2DUP @  SWAP 1 - !  SWAP @  COUNT 1 + ALLOT  !  147342  , ;
: :  CREATE  -1 ALLOT  SMUDGE  ] ;  forget

: CONSTANT ( n)  CREATE  -1 ALLOT  \ LITERAL  \ EXIT ;
: VARIABLE  CREATE  0 , ;
```

28

```
-SHORT checks if last instruction was a 5-bit literal.
FIX  merges 5-bit literal with new instruction.
SHORT  requires 5-bit literal (register, address or increment)
   for current instruction. Error message.

@  and  !  compile 5-bit or stack address instructions.
I@  and I!  compile 5-bit register instructions.
@+  and !+  compile 5-bit increment instructions.

PUSH  and  POP  push and pop the return stack.
   They are usually designated >R and R> .
I  copies the return stack onto the parameter stack.
TIMES  pushes the return stack to repeat the next instruction for
   n + 2 cycles.
```

29

```
OR,  compiles a 12-bit address with a backward jump instruction.
BFGIN  saves HERE for backward jumps.

UNTIL  compiles a conditional backward jump.
AGAIN  compiles an unconditional backward jump.
THEN  adds 12-bit current address into forward jump.
IF  compiles a conditional forward jump.
WHILE  compiles a conditional forward Jump - out of structure. ;
REPEAT  resolves a BEGIN ... WHILE ...  loop.
ELSE  inserts false clause in an IF ... THEN conditional.

FOR  compiles return stack push for a down-countinv loop.
NEXT  compiles a backward decrement-and-jump.
```

30

```
STRING  compiles a character string with a specified delimiter.

ABORT"  DOT"  are target versions of previously-defined host
   words.
(  skips over a comment. It must be defined in both FORTH and
   COMPILER .

RESET  restores dictionary to power-up status. It must be the
   last word in the dictionary.  It is called by  reset .

Insert application code before this block, to avoid using these
   Common target words. Alteratively,  FORGET  them.
```

28

```
  ( uCODE)  OCTAL
: -SHORT ( - t)  ?CODE @ @  177700 AND  157500 XOR ;
: FIX ( n)  ?CODE @ @  77 AND OR  ?CODE @ ! ;
: SHORT ( n)  -SHORT IF  DROP ABORT" n?"  THEN FIX ;

COMPILER
: @  -SHORT IF  167100 ,C  ELSE  147100  FIX THEN ;  forget
: !  -SHORT IF  177000 ,C  ELSE  157000  FIX THEN ;  forget
: I@  147300 SHORT ;
: I!  157200 SHORT ;
: @+  164700 SHORT ;
: !+  174700 SHORT ;

: R>  147321 ,C ;
: POP  147321 ,C ;       : PUSH  157201 ,C ;
: I  147301 ,C ;         : TIMES  157221 ,C ;  forget
```

29

```
  ( Structures)  OCTAL
FORTH { : OR, ( n n) }  \\  SWAP 7777 AND  OR ,.;
COMPILER : BEGIN ( - a)  H @  \\ ;

: UNTIL ( a)  110000 OR, ;
: AGAIN ( a)  130000 OR, ;
: THEN ( a)  \ BEGIN  7777 AND  SWAP +! ;
: IF ( - a)  \ BEGIN  110000 , ;
: WHILE ( a- a a)  \ IF  SWAP ;
: REPEAT ( a a)  \ AGAIN  \ THEN ;
: ELSE ( a - a)  \ BEGIN 130000 ,  SWAP  \ THEN ;

: FOR ( - a)  \ PUSH  \ BEGIN ;
: NEXT ( a)  120000 OR, ;
```

30

```
  ( Strings)  HEX
   FORTH : STRING ( n)  WORD @  7 TIMES 2/  1 + ALLOT ;

COMPILER : ABORT"  COMPILE abort"  22 STRING ;
         : ."  COMPILE dot"  22 STRING ;
         : (  29 WORD DROP ;

   FORTH : (  \ ( ;

: RESET  FORGET 0 ;  RECOVER  ` RESET dA @ -  ' reset !
```

Appendix B. Glossary of cmFORTH

a: address b: byte d: double integer f: flag n: integer t: true flag u: unsigned integer #: count Identifier nnX (nn screen number, X type code) C: target compiler F: FORTH I: COMPILER H: hidden V: variable

!	n a -	24F	Store n to memory at address a.
!	-	28I	Compile optimized store code.
!+	n -	28I	Compile increment store code.
!-	n -	28I	Compile decrement store code.
#	-	3C	Alias for end-of-line EXIT.
#	.. # n - .. #' n'	15F	Convert one digit from n and add it to output string.
##	a n - a a #	18H	Send serial disk read command to host computer.
#>	.. # n -	15F	Output number string to terminal.
#S	.. # n - .. #' 0	15F	Convert n and add digits to the output string.
'	- a	22F	Search dictionary for next word. Return code address.
(.)	n - .. #	15F	Convert n to an ASCII output string on stack.
*	n1 n2 - r	9F	Signed multiply of n1 and n2.
*'	-	7I	Compile multiply step code.
*-	-	7I	Compile signed multiply step code.
*/	n1 n2 u - r	9F	Ratio of n1xn2/u.
*/MOD	u1 u2 u3 - r q	9F	u1*u2/u3. Return remainder and quotient.
*F	-	7I	Compile fraction multiply step.
+	-	6C	Optimizing + code compiler.
+	n1 n2 - n3	24F	Add top two stack items.
+!	n a -	10F	Add n to memory at address a.
,	n -	25F	Compile n to top of dictionary.
,A	a -	25F	Compile address a to dictionary.
,C	n -	25F	Compile n as a machine code.
-	-	6C	Optimizing - code compiler.
-	n1 n2 - n3	24F	Subtract top from second stack item.
-'	- a t, p a f	22F	Search dictionary for next word. Return false if found.
-1	-	4C	Compile -1 code.
-DIGIT	b - n	21F	Convert character b to a digit.
-FIND	a1 a2 n - a1 t,a2 f	20H	Search dictionary for word at a1 with hash code n.
-LETTER	a1 a2 # - a2 a4	19H	Copy cell string at a1 to byte string at a2.
-M/MOD	d u - q r	9F	Divide d by u, and return remainder and quotient.
-SHORT	- f	28F	Return true if last compiled code has literal field.
.	n -	15F	Free format display of top stack item.
."	-	16F	Print the following text.
/	n u - q	9F	Divide by unsigned integer.
/'	-	7I	Compile divide step code.
/"	-	7I	Compile last divide step code.

/MOD	u1 u2 - r q	9F	Unsigned divide. Return remainder and quotient.
0+c	-	4C	Compile carry adjust code.
0<	-	1C	Fixed up 0< for prototype NC4000.
0<	-	6C	Optimizing 0< code compiler.
0<	n - f	24F	Return true if top stack item is negative.
0=	n - f	7F	Return true if top stack item is zero.
10*+	u1 b - u2	21H	Accumulate a digit b to product u1.
2!	d a -	10F	Store double integer to a.
2*	-	6C	Optimizing left shift code comiler.
2/	-	6C	Optimizing right shift code compiler.
2/MOD	n - rem quot	8F	Divide n by 2 and return remainder and quotient.
2@	a - d	10F	Fetch double integer at a.
2C@+	a - a+1 l h	10F	Increment a and return its contents in two bytes.
2DROP	d -	10F	Discard top two stack items.
2DUP	d - d d	11F	Duplicate top two stack items.
:	-	27F	Start a new colon definition.
;	-	5C	Optimizing ; compiler.
;	-	26I	Terminate a colon definition.
<	n1 n2 - f	7F	Return true if second item is less than top.
<#	n - ..# n	15F	Start a number output string.
=	n1 n2 - f	7F	Return true if top two items are equal.
>	n1 n2 - f	7F	Return true if second item is greater than top.
>IN	- a	12V	Pointer to input stream for text interpreter.
>R	-	28I	Compile >R code to retrieve from return stack.
?	a -	15F	Display contents of memory a.
?CODE	- a	12V	Pointer to address of code just compiled.
?DUP	n - n n, 0	11F	Duplicate n if it is not zero.
@	a - n	24F	Fetch contents of memory at address a.
@	-	28I	Compile smart fetch code.
@+	n -	28I	Compile increment fetch code.
@DROP	-	4C	Compile ÄROP code.
ABORT"	-	16F	Abort to text interpreter with an output message.
ABS	n - u	11F	Return absolute value of top stack item.
ABSENT	n - a	17H	Return buffer address a if block n is in a buffer.
ADDRESS	n - a	17H	Return buffer address of the nth disk block.
AGAIN	a -	29I	Compile an unconditional branch to address a.
ALLOT	n -	25F	Allocate n cells in the dictionary.
AND	-	6C	Optimizing AND code compiler.
AND	n1 n2 - n3	24F	AND top two stack items.
ARRAY	-	12H	Create a new array.
BASE	- a	12V	Number base for numeric I/O conversion.
BAUD	-	23H	Wait a B from terminal and determine the baud rate.
BEGIN	- a	29I	Starting point of a indefinite loop.
BINARY	n1 n2 -	6C	Defining word for optimizing ALU code compilers.
BLK	- a	12V	Contains the block number under interpretation.
BLOCK	n - a	18F	Read a block from host. Return buffer address a.

BUFFER	n - a	18F	Get a disk buffer for block n. Return buffer address.
BUFFERS	- a	12H	Array of block numbers of blocks in the disk buffers.
C!	b a -	10F	Store byte b to byte address a.
C/B	- a	12V	Machine cycles per bit for serial terminal.
C@	a - b	10F	Fetch a byte from byte address a.
CLIP	a -	2C	Relink the target vocabulary starting from a.
CNT	- a	12V	Count of characters received from terminal.
COMPILE	-	26F	Compile next address to dictionary.
COMPILER	-	4C	Switch context to COMPILER vocabulary.
CONSTANT	-	27F	Create a new constant.
CONTEXT	- a	12V	Context vocabulary hash code.
COUNT	n1 - n2	26H	Extract length from first cell in the name field.
CR	-	13F	Output a carriage return and a line feed.
CREATE	-	27F	Create a new definitions.
CURSOR	- a	12V	Pointer to the input character just received.
CYCLES	n -	11F	Run n empty for-next cycles.
D2*	-	7I	Compile double integer left shift code.
D2/	-	7I	Compile double integer right shift code.
DECIMAL	-	23F	Set base to 10 for decimal I/O.
DIGIT	n - b	15F	Convert n to a digit b.
DOES	-	26F	Define an inner interpreter.
DROP	n -	24F	Discard top item on stack.
DUMP	a - a+8	15F	Display 8 consecutive cells from address a.
DUP	n - n n	24F	Duplicate top stack item.
DUP?	-	4C	Pack previous DUP into current code if possible.
ELSE	a1 - a2	29I	Start a false clause in a branch structure.
EMIT	b -	13F	Send one byte to terminal.
EMPTY	-	1C	Starting point of a dictionary overlay.
EMPTY-BUFFERS	-	18F	Erase all the buffer pointers to empty buffers.
ERASE	a n -	10F	Zero n cells starting from a.
ESTABLISH	n a - a	17H	Identify oldest buffer with block n.
EXECUTE	a -	11F	Call subroutine at address a.
EXIT	-	5C	Optimizing EXIT compiler.
EXIT	-	26F	Return to the calling routine.
EXIT	-	26I	Compiler of EXIT machine code.
EXPECT	a n -	14F	Input a text string of n cells to address a.
FILL	a # n -	10F	Fill # cells of memory at a with n.
FIX	n -	28F	Insert 5 bit literal n into last compiled code.
FLUSH	-	18F	Write all updated buffer back to the host.
FOR	- a	29I	Start a definite loop.
FORTH	-	4C	Switch context to FORTH vocabulary.
H	- a	12V	Pointer to top of dictionary.
H'	- a	2C	Pointer to top of target dictionary.
HASH	n - a	20H	From hash code n return vocabulary link address a.
HERE	- a	14F	Return first free address on top of dictionary.

HEX	-	24F	Change number base to 16.
HOLD	.. # n b - .. # n	14F	Add byte b to the number output string on stack.
I	-	28I	Compile I code to copy from return stack.
I!	-	4C	Compile an optimized register store code.
I!	n -	28I	Compile internal register store code.
I@	n -	28I	Compile internal register fetch code.
I@!	n -	8C	Compile optimized register exchange code.
I@!	n -	28I	Compile registe exchange code.
IDENTIFY	n a - a	17F	Identify PREV buffer with block n.
IF	- a	29I	Start a conditional branch structure.
INTERPRET	n1 n2 -	22F	With BLK and >IN on stack, interpret text in input buffer.
KEY	- n	13F	Get one byte from terminal.
LETTER	a1 a2 n - a3 a4	19H	Copy cell string a1 to byte string at a2.
LITERAL	n -	25I	Compile n as a literal to dictionary.
LOAD	n -	24F	Interpret text in block n.
M*	n1 n2 - d	9F	Multiply n1 by n2 and return double integer product.
M*+	u1 0 u2 - d	9F	Unsigned multiply of u1 by u2.
M/	d u - q	9F	Divide d by u, and return quotient only.
M/MOD	ud u - q r	8F	Unsigned divide of ud by u. Return remainder and quotient.
MAX	n1 n2 - max	11F	Return the greater of n1 and n2.
MD	-	8C	Compile multiplier/divisor register code.
MIN	n1 n2 - min	11F	Return the smaller of n1 and n2.
MOD	u1 u2 - r	9F	Unsigned divide. Return remainder only.
MOVE	a1 a2 n -	10F	Move n+1 cells from a1 to a2-n.
MSG	- a	12V	Pointer to the terminal input buffer.
N!	-	4C	Compile local memory store code.
NB	- n	12F	Number of disk buffers less 1.
NEGATE	n1 - n2	24F	Negate top stack item.
NEXT	a -	29I	Terminate a definite loop.
NOP	-	4C	Compile NOP code.
NOT	n - f	7F	Return true if top stack item is false.
NUMBER	a - n	21F	Convert string at a to a number.
OCTAL	-	24F	Change number base to 8.
OFFSET	- a	12V	Offset for disk block 0.
OLDEST	- a	12V	Pointer to oldest disk buffer.
OR	-	6C	Optimizing OR code compiler.
OR	n1 n2 - n3	24F	OR top two stack items.
OR,	a n -	29H	OR address a to branch code n and compile it.
OVER	n1 n2 - n1 n2 n1	24F	Copy second item on stack.
PACK	a n - a	5C	Compile return code or pack return bit to last code.
PREV	- a	12V	Pointer to last referenced disk buffer.
PREVIOUS	- a n	26F	Return name field address and first cell in name field.
PRUNE	-	2C	Relink the target vocabulary.

QUIT	-	22F	The text interpreter.
R>	-	28I	Compile R> code to move to return stack.
R>DROP	-	4C	Compile R>DROP code.
RAM	- a	2C	Pointer to the top of RAM area for new variables.
RECOVER	-	2C	Recover one cell from compiled definition.
RECURSIVE	-	26I	Unsmudge the last definition.
REMEMBER	-	1C	Create a wall for dictionary overlay.
REPEAT	a1 a2 -	29I	Resolve the BEGIN-WHILE-REPEAT structure.
RESET	-	23F	Initialize the Forth system.
ROM	- a	23H	Array containing initial values of variables.
ROT	n1 n2 n3 - n2 n3 n1	7F	Rotate third stack item to top.
RX	- n	13F	Get one bit from terminal.
S'	-	7C	Compile square-root step code.
SAME	a1 a2 - a3 a4 f	20H	Search string at a1 through dictionary starting a2.
SCAN	a1 - a2	2C	Find the next host definition in linked chain.
SHIFT	n -	6C	Defining word for optimizing shift code compilers.
SHORT	n -	28F	Insert literal n into the last literal code.
SIGN	.. # n - .. #'	15F	Add - sign to output string if n is negative.
SMUDGE	-	26F	Set smudge bit in the last definition.
SPACE	-	14F	Output a space.
SPACES	n -	14F	Output n spaces.
SR	-	8C	Compile square-root register code.
SWAP	n1 n2 - n2 n1	24F	Exchange top two stack items.
SWAP-	-	6C	Optimizing SWAP- code compiler.
SWAP-DROP	-	4C	Compile NIP code.
THEN	a -	29I	Resovle conditional branching address.
THRU	n1 n2 -	1C	Load blocks from n1 to n2 inclusive.
TIMES	-	28I	Compile TIMES code for one instruction loop.
TRIM	a1 a2 - a3	2C	Relink the target and host vocabulary.
TWO	-	4C	Compile two cycle Nop.
TYPE	a1 - a2	13F	Output a stored string at a1. Return a2 after the string.
U*+	u1 r u2 - d	8F	Multiply u1 by u2 and add the product to r.
U.	u -	15F	Display unsigned integer u in free format.
U.R	u n -	15F	Display unsigned integer u right justified in n columns.
U<	u1 u2 - f	7F	Return true if second is less than top in unsigned comparison.
UNTIL	a -	29I	Compile a conditional branch to address a.
UPDATE	-	17F	Update the buffer pointed to by PREV.
UPDATED	- a n	17H	Return block number and address of oldest buffer.
USE	a -	26F	Install inner interpreter a to last definition.
VARIABLE	-	2C	Defining word for variables in target system.
VARIABLE	-	27F	Create a new variable.

VNEGATE	n1 n2 - n3 n4	9F	Negate top two stack items.
WHILE	a1 - a2 a1	29I	Compile branch code in an indefinite loop.
WIDTH	- a	12V	Cell size of name field.
WITHIN	n l h - f	11F	Return true if n is between l and h.
WORD	n - a	19F	Parse next word and move it to word buffer at a.
XOR	-	6C	Optimizing XOR code compiler.
XOR	n1 n2 - n3	24F	Exclusive OR of top two stack items.
[	-	25I	Stop compiling and start interpreting.
]	-	25F	Begin the compiler loop.
abort"	- n 0	16H	Runtime routine of ABORT".
begin	- a	29H	Mark current address for branching.
block	n a - n a	18H	Read a block from host and store it at address a.
buffer	n - a	18H	Return the buffer address of block n.
dA	- a	12V	Memory address offset for target compiler.
dot"	-	16H	Runtime routine of .".
uCODE	n -	4C	Defining word to create machine code compilers.
{	-	2C	Exchange pointers to Forth and target dictionaries.
}	-	2C	Exchange pointers to Forth and target dictionaries.
}	-	2C	Compile target dictionary address to host definition.

Index