JavaForthMachine

Computer Instruction Set

Hundreds or even thousands of computers were manufactured over the last century. Computer designs were characterized most clearly by the instruction set they executed. So far we knew lots of bad computer designs and bad computers filled a vast amount of our industrial waste dumps. Intel enshrined their X86 chips in its museum, but they will not be worshipped any soon now. MIPS and RISC came and went. ARM chips are in every cell phones. Which instruction set will be the best and to survive to the next century?

Looking around, I can see only one universal computer instruction set, and it is now gradually prevailing. It is Java. It was originally developed in the Sun Microsystems. After Sun failed in the marketplace to Intel and ARM, Oracle brought it up with the complete ownership of Java.

I knew Java was a close cousin of Forth. Until I start reading Java Virtual Machine Specifications. I didn't fully understand Java language and Java programming, particularly the object-oriented aspects of it. I went out and wrote an object-oriented-eForth system to run Forth on a webpage in Java. Then I realized how deep the relationship was. I saw a Forth Machine in the JVM specification, in spite that I still do not understand many of the JVM bytecode. From my humble opinion, the Java designers didn't fully understand the significance of the return stack, and adopted the concept of stack frames in the original RISC architecture. If we replaced the stack frames with a true return stack, you would get a simpler and much more efficient computer.

Since I do not fully understand Java and its JVM, I like to demonstration a Forth Machine based on the JVM bytecode, as much as possible. I will call it the JavaForthMachine or JFM. The goal of my demonstration will be a FPGA implementation of JFM on the QartusII IDE from Altera, now part of Intel. In 2005 I got a Altera NIOS FPGA kit with a Stratix II FPGA kit. With the then hot Quartus II IDE. I implemented a focal plane image processor and showed it to the NASA contractor. Nothing came out of this project, because the Stratix II was not big enough to do realistic image processing.

TheNIOS kit was on my shelf all theses years. The Don Golding set up an AI Robotic Group to promote a Forth FPGA chip for robots. He wanted the group to use the Lattice ICE40 FPGA chip because a very nice fpice40 kit was sellinf for $25 on Amazon. He also required that everybody program in SystemVerilog. My JFM didn't quite fit his requirements, as I had done all my chip designs in VHDL and even a helf decent JFM would not fit in an ICE40 chip.

I tried to reactivate my Quartus II license to not avail. Intel would not honor the expired license from its acquired Altera. However, Intel did grant me a no-cost Quartus beginner license to work with a non-specific Stratix II FPGA. I could not use my NIOS kit, because it required an obsolete PC with a serial UART port and a parallel printer port. These long

gone golden days in the Silicon Valley. With these restrictions, I set ny goals really low. I would only do an 32-bit integer JVM design in VHDL, and prove it only on the Quaetus II simulator. In the core, I am a software guy. It is better to let the hardware engineers to worry about the FPGA chips.

Then, what about Don's demands on SystemVerilog?

On Christmas 2021, my grandchildren decided to give me a $100 Amazon gift card. Kind of expansive gift from small kits. I thought long and hard to buy gifts worthy of their expectations. Final I decided it is about time that I learn a new language, and bought these cheap used books on Verilog and SystemVerilog, and started coverting my VHDL code to SystemVerilog.

I Googled to find side which can convert VHDL code to SystemVerilog code. Several sites on GitHub showed the converters, with 'make' builders. I hated make, and tried to avoid Linux all my life. There was a nice site offering on-line conversion. I pasted VDHL code in one windows, and Verilog code showed up in another window. When I tried save the Verilog file, the site complained I enter the wrong password. Can't save the Verilog file, period.

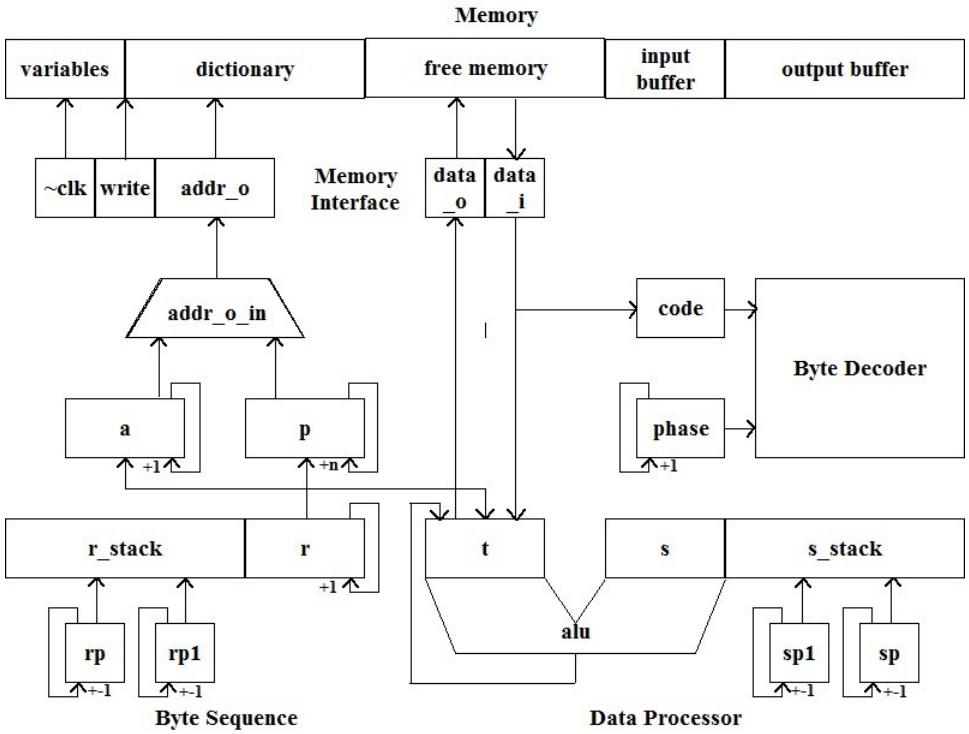
But, I saw how VHDL was translated to Verilog. Mostly the conversion was syntactic. I just had to follow the translation line by line with the text editor in Quartus II. Quartus II does support Verilog, and compiled my Verilog code. The most troubling and persistent warning I got was that some combinatorial signal implied latchs. I was not clear in the assignments of 'wires' and 'regs', and their groupings in th 'always' blocks.

Quartus II does not compile SystemVerilog file explicitly. It compiles only .v files, not .sv files. However, when I change Verilog reg and wrie to logic, Quartus threw out an error message that I was using a SystemVerilog2005 feature. Quartus II knew about SystemVerilog, but I didn't know how to invoke it. After a while I found that option in themenu item; configuration\language\SystenVerilog2005..

At this moment I had three JFM implementations, eJ32k.vhd in VHDL, eJs32k.v in SystemVerilog, all compiled, synthetized, and all simulated with the same outer interpreter produced by my metacompiler eJ32i.

Verilog, and eJsv32

JavaForthMachine



Registers in JFM

They are updated on the posedge of clk, if the corresponding regload flags are set.
The input data presented on the multiplexer reg_in is copied into the reg register.
The self-incrementing register p always point to the next program byte to be read from the byte memory instantiated as ram_memory module.
A byte data_i read from memory is latched into the code register, decoded, and sends out control signals to all parts of JFM to execute this bytecode.
If a multiple-byte bytecode is executed, the next phase is latched into the phase register.
Code and phase registers controls exactly how each byte is decoded.
The ram_memory contains data other than bytecode, using a data pointer register, a. a is autoincrementing, just like p, always pointing to the next data byte in ram_memory. It allows sequential data bytes to be read into different parts of the t register.
A valid memory address must always appear in the memory address input port addr_o. Either address from p or a register must be latched in addr_o_in and sent to addr_o. On a memory read operation, addr_o is latched in the ram_memory module on the negedge of clk, so that data_i will be available on the posedge of clk. On a memory write operation, addr_o and data_o are latched in the ram_memory module on the negedge of clk, so that data_o will be written to memory on the posedge of clk.

Memory designer do not know how to design a good memory chip. Designed like the RAMDQ module in the Megacore library, addr_o and data_0 must be latch first and data_i will be available shortly afterward necesiating a delay of one clock cycle. Memory chips should be design as asynchronous memory. Giving it a valid address and it must return its value asap. If writing, write_o must be available to be written on the next clock. There is absolutely no need of an extra clock cycle to latch addr_o and data_o.

P, a, addr_o, data_i, and addr_o_in registers are parts of the JFM bytecode sequence, to geter with the return stack r_stack. They support the sequencing of bytecode, nesting an unnesting of subroutines.

S_stack and r_stack are circular buffers 32 levels deep to stack up integer data and return addresses in JFM. The stacks do not overflow nor underflow. They do not need flushing or refilling. In the JFM outer interpreter, you wii always see the top 4 elements on s_stack. That all you need to know. You have no need to view the r_stack. You have to trust me that the stacks works in JFM.

The t register is the heart of this JFM. Data read from t register, s_stack, a register, and from ram_memory are processed in a data processor, and the results are written back to the t register. I used to code this data processor as a giant multiplexer feeding the t register. With JFM it is difficult to code the data processor explicitly, covering all available data paths. So I took the easier approach: feeing all data paths to a tristate buss t_in. On the rising edge of the master clock clk, whatever is active on the buss get latched into the t register if tload is a 1'b1. All other latching registers work the same way.

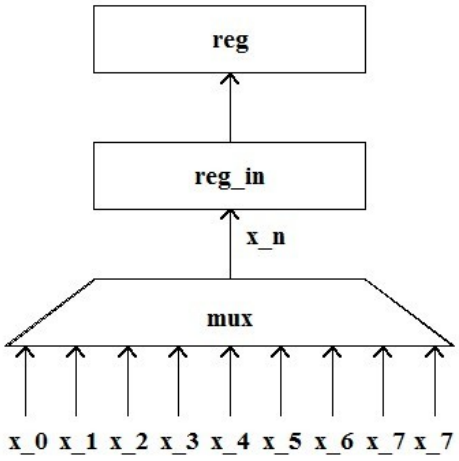
To operate a first-in-last-out stack correctly, you have t implement the mechanism to pre-

increment and post-decrement the stack pointer. Or you can do it the other way around: pre-decrement and post-increment. I took the pre-increment and post-decrement route, because it is easier to thin positively. A post decrement operation does not another clock cycle, because the stack pointer can be decremented at the end od a clock cycle. However, pre-incrementing the stack pointer requires an extra clock cycle. To save this extra cycle I use two stack pointers, sp and sp1 for s_stack. When doing a pre-incrementing, sp1 is used; otherwise, sp is used to select the proper stack element. Sp1=sp+1, and they are alwas in sync. No better way to implement a stack.

The bytecode decode generate all combinatorial signals to drive the JFM. It uses the code register to select a finite state machine paced by a phase number in the phase register. In each state, appropriate signals are sent to other parts of JFM and waiting for the next rising edge of clk.

Most bytecode are finishd in a single clock. However. Multi-byte code need more clock cycles to complete. Phase register auto-increments to carry through all phases. In the last phase, phase register is cleared to 0 to execute the next bytecode fetched from the next ram_memory.

reg: registers p, a, t, sp, sp1, rp, rp1, code, phase, s_stack, r_rstack
reg_in: input_reg
regload: load flag
clk: posedge clock



Metacompiler eJ32i.fex

The metacompiler was used to produce a memory image for the JavaForthMachine to execute. The memory image consist of a Forth dictionary, system variables to run the Forth outer interpreter, input and output text buffers. The dictionary contains word records of al primitive words, with Java Bytecode in their code fields. eJ32i.fex is a Forth (F#) project file build and testing the JFM.

```
( eJ32i, 13dec21cht, Java bytecode, input, output)
( eJ32h, 29nov21cht, Java bytecode, >r,r@, >r)
( eJ32g, 17nov21cht, Java bytecode, case, top)
( eP32s, 23jun21cht, bytecode, subrouting threading )
( JFM_56, 06MAR19cht )
\ HTTP server
( JFM_54, 02MAR19cht )
\ HTTP server
( JFM_53, 19feb19cht )
\ load LOAD.TXT from flash on boot
( ep32r, move to F#, 6/3/2021 cht )
( copy ep32q to JFM_50 )
( JFM_52, 22jan19cht )
\ fugues and musete
( JFM_51, 21jan19cht )
\ add peeks.txt and organ1.txt
( JFM_50, 15jan19cht )
\ Move from JFMto ESP32, for AIR robot
\ cEFa  10sep09cht
\ Goal is to produce a dictionary file to be compiled by a C
compiler
\ Assume 31 eForth primitives are coded in C
\ Each FORTH word contains a link field, a name field, a code
```

```
field
\    and a parameter field, as in standard eForth model
\ The code field contains a token pointing to a primitive word
\ Low level primitive FORTH words has 1 cell of code field
\ High level FORTH word has call in code field and a address
list
\ Variable has doVAR in code field and a cell for value
\ Array is same as variable, but with many cells in parameter
field
\ User variable has doUSE in code, and an offset as parameter
\
\

 FLOAD init.f         \ initial stuff
 FLOAD win32.f        \ win32 system interface
 FLOAD consolei.f     \ api and constant defination

 FLOAD ui.f           \ user interface helper routine
( reposition )

 FLOAD console.f      \ the main program
 FLOAD ansi.f
 FLOAD fileinc.f
 FLOAD META32i.f

cr .( Version FIX eJ32 forth )
```

META32j.f

Metacompiler is a term used by Forth programmers to describe the process of building a new Forth system on an existing Forth system. The new Forth system may run on the same platform as the old Forth system. It may be targeted to a new platform, or to a new CPU. The new Forth system may share a large portion of Forth code with the old system, hence the term metacompilation. In a sense, a metacompiler is very similar to a conventional cross assembler/compiler.

The JFM metacompiler is contained in file META32j.f. It allocates a data array ram, and deposits records of primitive commands and compound commands to build a dictionary for Forth. The JavaForthMachine (JFM) was programmed in eJ32k.vhd in VHDL, eJv32k in Verilog, and in eJev32k.v in SystemVerilog. All functions represented by sets of Java Bytecode. These bytecode are assembled into the code fields of primitive commands. The addresses of code fields are compiled into goto lists in the code fields of compound commands

Meta32j.f file carries out the first step of metacompilation. It declareds a big memory ararry RAM to host the dictionary of JavaForthMachine. Java Bytecode and other information are deposit in to the RAM array with RAMC! (for bytes), RAMW! (for shorts), and RAMC@ (for integers). RAMC@ (for bytes), RAMC@ (for shorts), and RAM! (for integers). RAMC@ (for bytes), RAMW@ (for shorts), and RAMC! (for integers) will be used to retrieve information from the RAM array. Memory addresses are al byte addresses.

After setting up the environment to build a target dictionary, META32i.f loads source code from three other files to do specific jobs:

`ASM32i.f`	JFM assembler
`KERN32i.f`	Assemble primitive commands
`EF32i.f`	Compile compound commands

Source code in META32i.f is lengthy, and it is best to comment each command to bring out its function and meaning.

`debugging`? (-- a) A variable containing a switch to turn break points on and off. When debugging? is set to -1, compilation will stop and the data stack is dumped when a “cr” command is executed. Sprinkling “cr” commands in the source code file allows you to watch the progress of metacompilation and even stops it when necessary.

```
variable debugging?
\ -1 debugging? !
```

`.head` (a – a) Display name of a command that is about to be compiled. It is used to display a symbol table. You can look up the code field address of any command in this table.

```
: .head ( addr -- addr )
   SPACE >IN @ 20 WORD COUNT TYPE >IN !
   DUP .
   ;
```

`cr` (--) Stop metacompilation if debugging? is -1, and dump data stack. If you press control-A, metacompilation is aborted. Otherwise, metacompilation continues. It is a NOP if debugging? is 0.

```
: CR CR
   debugging? @
   IF .S KEY 0D = IF ." DONE" QUIT THEN
   THEN
   ;
```

`break` (--) Pause metacompilation and dump data stack. If you press Return, metacompilation is aborted. Otherwise, metacompilation continues. It sets a break point.

```
: BREAK CR
   .S KEY 0D = IF ." DONE" QUIT THEN
   ;
```

During metacompilation, Forth commands will be redefined so that they compile tokens or assemble byte code into the target dictionary. There are numerous occasions where the original behavior of a Forth command must be exercised. To preserve the original behavior of a Forth command, it is assigned a different name. Thereby after a command is redefined, we can still exercise its original behavior by invoking the alternate name.

For example, "+" is a Forth command that adds the top two numbers on the data stack in the F# system. Then in the cefKERNai.f file, a new "DUP" command is defined to assemble a dup, instruction in the target JFM system. If you still need to duplicate a number, you must use the alternate command "forth_dup" as shown below. All the F# commands you need to use later must be redefined as "forth_xxx" commands. If you neglect to redefine them, you will find that the system behaves very strangely.

```
: forth_dup DUP ;
: forth_drop DROP ;
: forth_over OVER ;
: forth_swap SWAP ;
: forth_@ @ ;
: CRR cr ;
```

The JFM executes commands and accesses data in the dictionary, range 0-1FFF. In F# we allocate a 8k byte memory array, "ram", to hold the JFM target dictionary. This array contains code and data to be copied into JFM data[] arry, to be executed on the JFMchip.

RAM (-- a) Memory array in F# for the JFM target dictionary. It has a logical base address of 0 for the JFM. Commands and data words in the target are stored in this array.

```
CREATE ram  8000 ALLOT
```

RESET (--) Clear "ram" image array, preparing it to receive code and data for the JFM.

```
: RESET   ram 8000 0 FILL ;   RESET
```

RAM@ (a – n) Replace a logical address on stack with data stored in "ram" dictionary.

```
: RAM@    ram +  @ ;
```

RAMC@ (a – c) Replace a logical address on stack with byte data stored in "ram" dictionary.

```
: RAMC@   ram +  C@ ;
```

RAM! (a n --) Store second integer on stack into logical address of "ram" dictionary.

```
: RAM!    ram +  ! ;
```

RAM! (a c --) Store second byte on stack into logical address of "ram" dictionary.

```
: RAMC!   ram +  C! ;
```

FOUR (a --) Display four consecutive words in target dictionary.

```
: FOUR   (  a -- ) 4 FOR AFT  DUP RAM@ 9 U.R  4 + THEN NEXT  ;
```

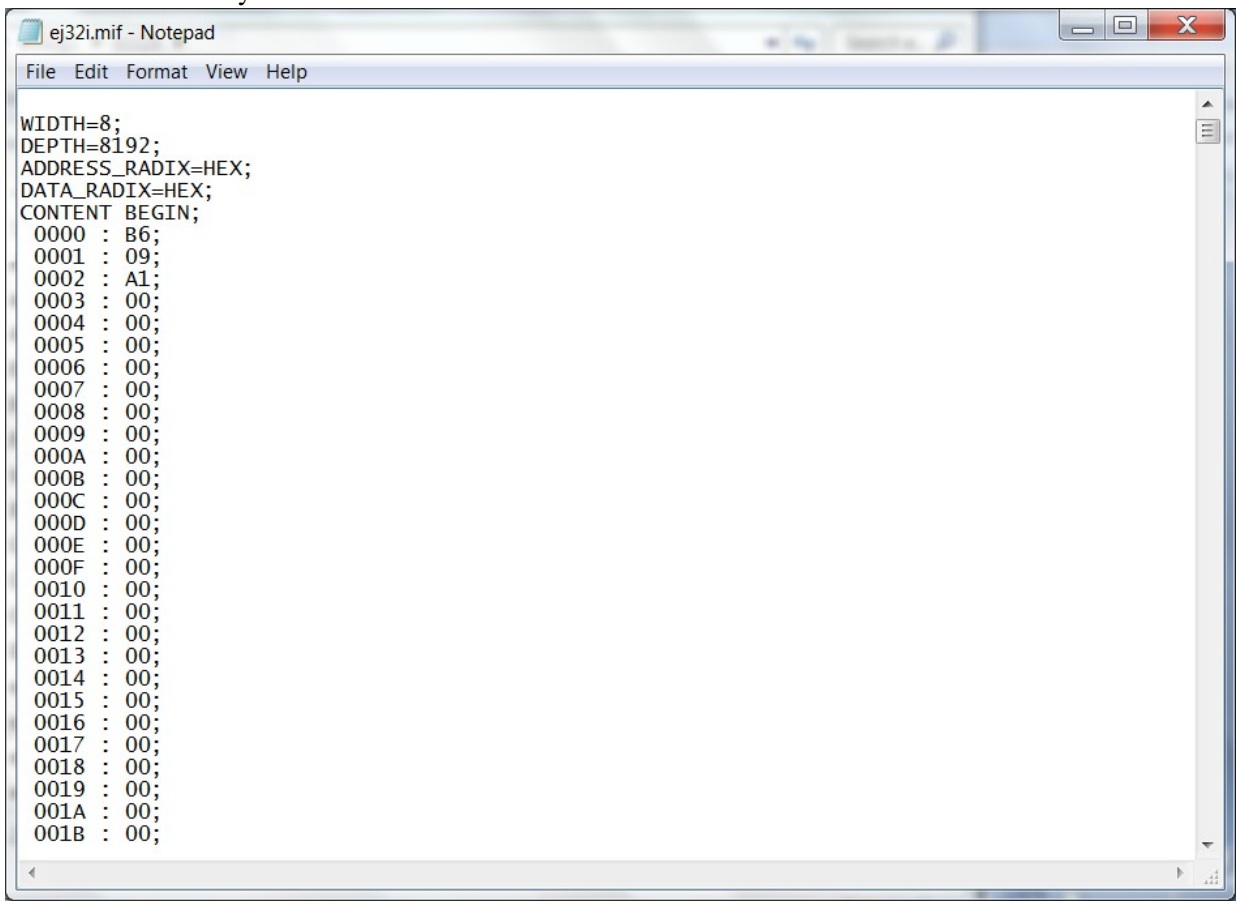
SHOW (a – a+128) Display 128 words in target from address "a". It also returns a+128 to "show" the next block of 128 words.

```
: SHOW ( a)   10 FOR AFT  CR  DUP 7 .R SPACE
      FOUR SPACE FOUR  THEN NEXT ;
```

SHOWRAM (--) Display the entire JFM dictionary of 2K words.

```
: showram 0 0C FOR AFT SHOW THEN NEXT DROP ;
```

The JFM metacompiler builds a target dictionary for the JFMchip in “ram. This dictionary eventually will be imported to the JFM_44.ino so that this dictionary will be incorporated in JFM. Arduino IDE requires that the dictionary be written in a file conforming to its long data[] array format, which consists of a header with a body containing memory information in hexadecimal numbers. The header and first few lines of the body are as follows:



This dictionary is written to a text file eJ32i.mif. Here are the commands to open this file, writing data to it, and closing it.

hFile (-- handle) A variable holding a file handle.

```
VARIABLE hFile
```

CRLF-ARRAY (-- a) A byte array containing CR and LF characters.

```
CREATE CRLF-ARRAY 0D C, 0A C,
```

CRLF (--) Insert a carriage return and a line feed into the currently opened file.
open-mif-file (--) Open a file named eJ32i.mif for writing.
write-mif-header (--) Write a header required by Arduino into current file.
write-mif-trailer (--) Write last line of text into current file.

`write-mif-data ( -- )` Write a 4K word image of the JFM dictionary from memory array "ram" to the eJ32i.mif file.
`close-mif-file ( -- )` Close eJ32i.mif file.
`write-mif-file ( -- )` open eJ32i.mif file, write a header, write data, write trailer, and then closes the file. eJ32i.mif containing 4K words of the JFM dictionary.

The JFM metacompiler continues to load the byte code assembler in cefASMi.f. In the assembler, all byte code of VFM are defined, and the ways they are assembled into code fields of primitive commands. Means to compile link fields and name fields to form headers of commands are also defined. It is now almost ready to assemble primitive commands for JFM.

```
CR .( include assembler )
FLOAD fASM32i.f
```

After the assembler is built, we are ready to build the kernel part of JFM dictionary. All primitive commands are assembled. The kernel starts at location 0x200, leaving rooms for the Terminal Input Buffer TIB in the area 0-0x17F. System variables from 0x180-0x1FF.

```
$200 ORG
CR .( include kernel )
FLOAD cefKERN32i.f
```

With the kernel in place, high level compound commands are compiled immediately after the kernel, by loading cEFai.f. The top JFM dictionary is at 0x1DAC so far. It is pushed on data stack by the commands '`H forth_@`', to be used later to initialize the system variable CP. With 4096 words allocated in data[] array, the space is about half full. You can compile substantial application in this dictionary. If you need more space, just allocate a bigger array.

```
CRR .( include eforth )
FLOAD cEF32i.f
H forth_@
```

ASM32i.f, KERN32i.f, and cEF32.f files will be discussed in separate chapters. Finally, several system variables must be initialized properly so that the Forth interpreter can work properly on boot.

At last, write the contents of JVM dictionary to `eJ32i.mif.`

```
write-mif-file
```

Done.

```
( meta32g.f for eJ32g, 17nov21cht )

HEX
VARIABLE debugging?
\ 1 debugging? !

: .head ( addr -- addr )
   SPACE >IN @ 20 WORD COUNT TYPE >IN !
   DUP .
   ;

: cr CR
   debugging? @
   IF .S KEY 0D = ABORT" DONE"
   THEN
   ;

: forth_' ' ;
: forth_dup DUP ;
: forth_drop DROP ;
: forth_over OVER ;
: forth_swap SWAP ;
: forth_@ @ ;
: forth_! ! ;
: forth_and AND ;
: forth_+ + ;
: forth_- - ;
: forth_word WORD ;
: forth_words WORDS ;
: forth_.s .S ;
: CRR cr ;
: forth_.( [COMPILE] .( ;
: forth_count COUNT ;
: forth_r> R> ;
: -or   XOR ;
: >body 5 + ;
: forth_forget FORGET ;

CREATE ram  2000 ALLOT
: reset  ram 2000 0 FILL ;
: ram@   ram +  count >r count >r count >r c@ r> r> r>
         8 lshift + 8 lshift + 8 lshift +  ;
: ram!   ram +  2dup 3 + c! swap 8 rshift swap 2dup 2+ c!
         swap 8 rshift swap 2dup 1+ c!
         swap 8 rshift swap c! ;
: ramw@  ram +  count 8 lshift swap c@ + ;
: ramw!  ram +  2dup 1+ c! swap 8 rshift swap c! ;
```

```
: ramc@  ram +  c@ ;
: ramc!  ram +  c! ;
: binary 2 BASE ! ;
: FOUR   (  a -- a+16 ) 10 FOR AFT  DUP RAMC@ 3 U.R  1 + THEN
NEXT
      10 - SPACE 10 FOR AFT  DUP RAMC@ 20 MAX 7E MIN EMIT 1 +
THEN NEXT ;
: SHOW ( a -- a+256 )   10 FOR AFT  CR  DUP 7 .R SPACE
      FOUR THEN NEXT ;
: showram 0 $E FOR AFT SHOW THEN NEXT DROP ;

VARIABLE hFile
CREATE CRLF-ARRAY 0D C, 0A C,
: CRLF
      hFile @
      CRLF-ARRAY 2
      PAD ( lpWrittenBytes )
      0 ( lpOverlapped )
      WriteFile
      IF ELSE ." write error" QUIT THEN
   ;

: open-mif-file
   Z" ej32i.mif"
   $40000000 ( GENERIC_WRITE )
   0 ( share mode )
   0 ( security attribute )
   2 ( CREATE_ALWAYS )
   $80 ( FILE_ATTRIBUTE_NORMAL )
   0 ( hTemplateFile )
   CreateFileA hFile !
   ;
: write-mif-line
      PAD ( lpWrittenBytes )
      0 ( lpOverlapped )
      WriteFile
      IF ELSE ." write error" QUIT THEN
   CRLF
   ;
: write-mif-header
   CRLF
      hFile @
      $" WIDTH=8;"
      PAD ( lpWrittenBytes )
      0 ( lpOverlapped )
      WriteFile
      IF ELSE ." write error" QUIT THEN
```

```
   CRLF
      hFile @
      $" DEPTH=8192;"
      PAD ( lpWrittenBytes )
      0 ( lpOverlapped )
      WriteFile
      IF ELSE ." write error" QUIT THEN
   CRLF
      hFile @
      $" ADDRESS_RADIX=HEX;"
      PAD ( lpWrittenBytes )
      0 ( lpOverlapped )
      WriteFile
      IF ELSE ." write error" QUIT THEN
   CRLF
      hFile @
      $" DATA_RADIX=HEX;"
      PAD ( lpWrittenBytes )
      0 ( lpOverlapped )
      WriteFile
      IF ELSE ." write error" QUIT THEN
   CRLF
      hFile @
      $" CONTENT BEGIN;"
      PAD ( lpWrittenBytes )
      0 ( lpOverlapped )
      WriteFile
      IF ELSE ." write error" QUIT THEN
   ;

: write-mif-data
   0 ( initial ram location )
   $2000 FOR AFT
      CRLF
      hFile @
      OVER
      <# 3A HOLD 20 HOLD 3 FOR # NEXT 20 HOLD #>
      PAD ( lpWrittenBytes )
      0 ( lpOverlapped )
      WriteFile
      IF ELSE ." write error" QUIT THEN
      hFile @
      OVER ramc@
      <# 3B HOLD # # 20 HOLD #>
      PAD ( lpWrittenBytes )
      0 ( lpOverlapped )
      WriteFile
```

```
      IF ELSE ." write error" QUIT THEN
      1+
   THEN NEXT
   DROP ( discard ram location )
   ;

: close-mif-file
   CRLF
      hFile @
      $" END;"
      PAD ( lpWrittenBytes )
      0 ( lpOverlapped )
      WriteFile
      IF ELSE ." write error" QUIT THEN
   CRLF
   hFile @ CloseHandle DROP
   ;

: write-mif-file
   open-mif-file
   write-mif-header
   write-mif-data
   close-mif-file
   ;

VARIABLE tests
VARIABLE tests-addr
VARIABLE tests-len
VARIABLE tests-end
VARIABLE tests-match
VARIABLE hFiletests
VARIABLE hMaptests
VARIABLE testsFileLength 0 ,
VARIABLE hFileEXT
VARIABLE EXTpointer ( extgp string at PAD )
VARIABLE EXTlength
VARIABLE WritenLength

20 constant tests-limit
0 tests-match !

: testsopen
  z" tests.txt"
  GENERIC_READ GENERIC_WRITE OR
  FILE_SHARE_READ
  0
  OPEN_EXISTING
```

```
  FILE_ATTRIBUTE_ARCHIVE
  0
  CreateFileA hFiletests !

  hFiletests @
  testsFileLength cell+
  GetFileSize testsFileLength !

  hFiletests @
  0
  PAGE_READWRITE
  0
  0
  0
  CreateFileMappingA hMaptests !

  hMaptests @
  FILE_MAP_READ FILE_MAP_WRITE OR
  0  \ file offset high
  0  \ file offset low
  0  \ #byte to map 0 = all
  MapViewOfFile tests !
;

: testsclose
  tests @ UnmapViewOfFile DROP
  hMaptests @ CloseHandle DROP
  hFiletests @ CloseHandle DROP
;

 testsopen
 tests @ ram $1000 + testsFileLength @ cmove
 testsclose

FLOAD asm32i.f
$100 org
FLOAD kern32i.f
FLOAD ef32i.f
write-mif-file
FLOAD sim32i.f
```

ASM32i.f Assembler

Byte Code Assembler

The ASM32i.f file contains a byte code assembler for JFM. It packs up to 5 byte code into one Java assembly program word. It first clears a program location pointed to by a variable “hw”. Assembly commands are executed to insert byte code into consecutive bytes, from right to left in the big endian order. Assembly commands make necessary decisions as to whether to add more byte code to the current program word, or start a new program word.

JFM has variable length byte code. Byte code are executed from right to left. Assembly commands for single byte code are defined by a defining word INST. Two byte code are defined by a defining word INSB. Three byte code are defined by a defining word INSW. 5 byte code are defined by a defining word INSI. Defining words in Forth makes this optimizing assembler very simple and very efficient.

The JFM system is based on a Subroutine Threading Model, in which a primitive command has byte code in its code field. To assemble a primitive command, the assemble first build a header, with a link field and a name field. After that, the assembler simply pack consecutive bytes with byte code until the primitive commands is completed.

`h ( -- a )` A variable pointing to the next free memory cell at the top of the target dictionary.
`VARIABLE h`
`lastH ( -- a )` A variable pointing to the name field of the current target command under construction.
`NAMER! ( d -- )` Compile an integer value, “d”, to the top of the target dictionary.
`COMPILE-ONLY ( -- )` Patch Bit 6 in first word of name field in current target command. Text interpreter checks it to avoid executing compiler commands.
`IMMEDIATE ( -- )` Patch Bit 7 in first word of name field in current target command. Compiler checks it to execute commands while compiling.
`ORG ( a -- )` Initialize pointer “h” to a new address to start assembling.
`#, ( d -- )` Compile an integer d to top of target dictionary. It is the most primitive assembler and compiler. The JFM assembler is an extension of this primitive assembly command.
`,W ( d -- )` Compile a short w to the program word pointed to by h. It generally fills the address field in the current byte code.
`,I ( d -- )` Compile a Java bytecode to dictionary.
`INST ( b -- )` Define an one byte bytecode assembly commands. It creates a byte code assembly command like a constant. When a byte code assembly command is later executed, this byte “b” is retrieved and a byte code is assembled into the current dictionary.
`INST ( b -- )` Define an one byte bytecode assembly commands. It creates a byte code assembly command like a constant. When a byte code assembly command is later executed, this byte “b” is retrieved and a byte code is assembled into the current dictionary.
`INSC ( b b -- )` Define an one byte bytecod with one additional data byte.

`INSW ( b w -- )` Define an one byte bytecod with 2 additional data byte.
`INSI ( b i -- )` Define an one byte bytecode with 4 additional data byte.
`0 INST nop,`

All JVM bytecode are then defined with a few more bytecode needed by the JFM.

Command Headers

In JFM, all primitive Forth commands are compiled in a target dictionary, and linked as a list. Each command has a link field of one 16-bit word, a variable length name field in which the first byte contains a length followed by the ASCII code of the name, a variable code field. A primitive command has byte code in its code field. A compound command has abyte code in its code field, and a token list in its parameter field. Here are commands to build headers, which include link and name fields.

`(makehead) ( -- )` Build a header for a new target command. The header includes a link field and a name field. The address of the name field in the last target command is stored in "`lasth`", and is compiled into the link field. "`h`" points to the name field of the new command, and is copied into "`lasth`". Now, the following string is packed into the name field, starting with its length byte, and null filled to the word boundary. Now, "`h`" points to the code field of this new target command.
`makehead ( -- )` Build a header with (`makehead`) and save the name string to define a compiler command in metacompiler. It displays the name and code field address. A string can be used repeatedly by saving and restoring its pointer in "`>IN`".
`$LIT ( -- )` Compile a packed string for a string literal inside a token list. It works similarly as (`makehead`). However, the name string is delimited by space character (ASCII 0x20), while a string literal is delimited by a double-quote character (ASCII 0x22).

Structured Assmbly

The JFM assembler can be pushed further to assemble short branches and loops for s number of more sophisticated JFM prmimitives.

`ldb` assembles byte literals.
`ldw` assembles short literals.
`LIT` assembles integer literals. May be changed later to byte literals.

`begin` implements begin-again, begin-until, begin-while-repeat.
`bz` implements if, until, and while.
`Ifeq` implements if, until, and while.
`bra` implements else and repeat.
`jmp` implements else and repeat.
`if` implements if.
`ifeqq` implements >.
`ifneg` implements 0<.
`ifgreat` implements >.

`ifless` implements <.
`skip` implements `bra.`
`else` implements `skip.`
`until` implements `bz.`
`while` implements `if.`
`repeat` implements `bra.`
`again` implements `bra.`
`aft ( a -- a' a" )` implements `skip.`
`for ( -- a )` implements `pushr.`
`next ( a -- )` implements `donext.`

Compilers for Primitive and Compound Commands

We are now at the peak of our metacompiler. We built all the tools to compile new Forth commands into the target dictionary, which will eventually run JFMchip. All commands have a link field and a name field. Primitive commands have an additional code field. Compound commands have a code field with token lists. Tokens are three bytes long, and are subroutine call bytecode. In JFM, bytecode and token can be intermixed freely. Two defining commands are now created to build the primitive and compound commands. `CODE` creates a header for a primitive commands, and its following code field can now be packed with byte code. `::` `(colon-colon)` creates a header for a compound command, and its following parameter field can be stuffed with a token list. `CODE` and `::` are defined identically though.

`CODE ( -- )` Create a new primitive command in JFM target dictionary. It creates a new header with a link field and a name field, and is ready to assemble byte code in the following code field. It also creates an assembly command in the metacompiler, storing its code field address. When this assembly command is encountered by metacompiler, it compiles its code field address as a token to extend the token list currently under construction.
`:: ( -- )` Create a new compound command in JFM target dictionary. It creates a link field and a name field, and then is ready to compile a new token list. Now, a token list is built in the code field, to become a new compound command in target dictionary. It also creates an assembly command in the metacompiler, storing its code field address. When this assembly command is encountered by metacompiler, it compiles its code field address as a token to extend the token list currently under construction.

```
\ asm32s.f 24jun21cht, bytecode, subroutine threading

HEX
VARIABLE h
VARIABLE lastH 0 lastH !   \ init linkfield address lfa

: nameR! ( d -- )
   h @ ramw!               \ store short to dictionar
   2 h +!                  \ bump h
   ;

: compile-only 40 lastH @ RAMC@ XOR lastH @ RAMC! ;
: IMMEDIATE    80 lastH @ RAMC@ XOR lastH @ RAMC! ;

: ORG   DUP . CR h ! ;
: #,   ( d ) H @ RAM!   4 h +! ;
: w,   ( d ) H @ RAMW!  2 h +! ;
: I,   ( d ) H @ RAMC!  1 h +! ;

: INST CONSTANT DOES> R> @ I, ;
: INSB CONSTANT DOES> R> @ I, I, ;
: INSW CONSTANT DOES> R> @ I, w, ;
: INSL CONSTANT DOES> R> @ I, #, ;

DECIMAL
00 ( 0x00  ) INST nop
01 ( 0x01  ) INST aconst_null
02 ( 0x02  ) INST iconst_m1
03 ( 0x03  ) INST iconst_0
04 ( 0x04  ) INST iconst_1
05 ( 0x05  ) INST iconst_2
06 ( 0x06  ) INST iconst_3
07 ( 0x07  ) INST iconst_4
08 ( 0x08  ) INST iconst_5
09 ( 0x09  ) INST lconst_0
10 ( 0x0a  ) INST lconst_1
11 ( 0x0b  ) INST fconst_0
12 ( 0x0c  ) INST fconst_1
13 ( 0x0d  ) INST fconst_2
14 ( 0x0e  ) INST dconst_0
15 ( 0x0f  ) INST dconst_1
16 ( 0x10  ) INSB bipush
17 ( 0x11  ) INSW sipush
18 ( 0x12  ) INSB ldc
19 ( 0x13  ) INSW ldc_w
20 ( 0x14  ) INSW ldc2_w
21 ( 0x15  ) INSB iload
22 ( 0x16  ) INSB lload
23 ( 0x17  ) INSB fload
24 ( 0x18  ) INSB dload
25 ( 0x19  ) INSB aload
26 ( 0x1a  ) INST iload_0
27 ( 0x1b  ) INST iload_1
28 ( 0x1c  ) INST iload_2
```

```
29 ( 0x1d  ) INST iload_3
30 ( 0x1e  ) INST lload_0
31 ( 0x1f  ) INST lload_1
32 ( 0x20  ) INST lload_2
33 ( 0x21  ) INST lload_3
34 ( 0x22  ) INST fload_0
35 ( 0x23  ) INST fload_1
36 ( 0x24  ) INST fload_2
37 ( 0x25  ) INST fload_3
38 ( 0x26  ) INST dload_0
39 ( 0x27  ) INST dload_1
40 ( 0x28  ) INST dload_2
41 ( 0x29  ) INST dload_3
42 ( 0x2a  ) INST aload_0
43 ( 0x2b  ) INST aload_1
44 ( 0x2c  ) INST aload_2
45 ( 0x2d  ) INST aload_3
46 ( 0x2e  ) INST iaload
47 ( 0x2f  ) INST laload
48 ( 0x30  ) INST faload
49 ( 0x31  ) INST daload
50 ( 0x32  ) INST aaload
51 ( 0x33  ) INST baload
52 ( 0x34  ) INST caload
53 ( 0x35  ) INST saload
54 ( 0x36  ) INSB istore
55 ( 0x37  ) INSB lstore
56 ( 0x38  ) INSB fstore
57 ( 0x39  ) INSB dstore
58 ( 0x3a  ) INSB astore
59 ( 0x3b  ) INST istore_0
60 ( 0x3c  ) INST istore_1
61 ( 0x3d  ) INST istore_2
62 ( 0x3e  ) INST istore_3
63 ( 0x3f  ) INST lstore_0
64 ( 0x40  ) INST lstore_1
65 ( 0x41  ) INST lstore_2
66 ( 0x42  ) INST lstore_3
67 ( 0x43  ) INST fstore_0
68 ( 0x44  ) INST fstore_1
69 ( 0x45  ) INST fstore_2
70 ( 0x46  ) INST fstore_3
71 ( 0x47  ) INST dstore_0
72 ( 0x48  ) INST dstore_1
73 ( 0x49  ) INST dstore_2
74 ( 0x4a  ) INST dstore_3
75 ( 0x4b  ) INST astore_0
76 ( 0x4c  ) INST astore_1
77 ( 0x4d  ) INST astore_2
78 ( 0x4e  ) INST astore_3
79 ( 0x4f  ) INST iastore
80 ( 0x50  ) INST lastore
81 ( 0x51  ) INST fastore
82 ( 0x52  ) INST dastore
```

```
83 ( 0x53  ) INST aastore
84 ( 0x54  ) INST bastore
85 ( 0x55  ) INST castore
86 ( 0x56  ) INST sastore
87 ( 0x57  ) INST pop
88 ( 0x58  ) INST pop2
89 ( 0x59  ) INST dup
90 ( 0x5a  ) INST dup_x1
91 ( 0x5b  ) INST dup_x2
92 ( 0x5c  ) INST dup2
93 ( 0x5d  ) INST dup2_x1
94 ( 0x5e  ) INST dup2_x2
95 ( 0x5f  ) INST swap
96 ( 0x60  ) INST iadd
97 ( 0x61  ) INST ladd
98 ( 0x62  ) INST fadd
99 ( 0x63  ) INST dadd
100 ( 0x64  ) INST isub
101 ( 0x65  ) INST lsub
102 ( 0x66  ) INST fsub
103 ( 0x67  ) INST dsub
104 ( 0x68  ) INST imul
105 ( 0x69  ) INST lmul
106 ( 0x6a  ) INST fmul
107 ( 0x6b  ) INST dmul
108 ( 0x6c  ) INST idiv
109 ( 0x6d  ) INST ldiv
110 ( 0x6e  ) INST fdiv
111 ( 0x6f  ) INST ddiv
112 ( 0x70  ) INST irem
113 ( 0x71  ) INST lrem
114 ( 0x72  ) INST frem
115 ( 0x73  ) INST drem
116 ( 0x74  ) INST ineg
117 ( 0x75  ) INST lneg
118 ( 0x76  ) INST fneg
119 ( 0x77  ) INST dneg
120 ( 0x78  ) INST ishl
121 ( 0x79  ) INST lshl
122 ( 0x7a  ) INST ishr
123 ( 0x7b  ) INST lshr
124 ( 0x7c  ) INST iushr
125 ( 0x7d  ) INST lushr
126 ( 0x7e  ) INST iand
127 ( 0x7f  ) INST land
128 ( 0x80  ) INST ior
129 ( 0x81  ) INST lor
130 ( 0x82  ) INST ixor
131 ( 0x83  ) INST lxor
132 ( 0x84  ) INSW iinc
133 ( 0x85  ) INST i2l
134 ( 0x86  ) INST i2f
135 ( 0x87  ) INST i2d
136 ( 0x88  ) INST l2i
```

```
137 ( 0x89  ) INST l2f
138 ( 0x8a  ) INST l2d
139 ( 0x8b  ) INST f2i
140 ( 0x8c  ) INST f2l
141 ( 0x8d  ) INST f2d
142 ( 0x8e  ) INST d2i
143 ( 0x8f  ) INST d2l
144 ( 0x90  ) INST d2f
145 ( 0x91  ) INST i2b
146 ( 0x92  ) INST i2c
147 ( 0x93  ) INST i2s
148 ( 0x94  ) INST lcmp
149 ( 0x95  ) INST fcmpl
150 ( 0x96  ) INST fcmpg
151 ( 0x97  ) INST dcmpl
152 ( 0x98  ) INST dcmpg
153 ( 0x99  ) INSW ifeq
154 ( 0x9a  ) INSW ifne
155 ( 0x9b  ) INSW iflt
156 ( 0x9c  ) INSW ifge
157 ( 0x9d  ) INSW ifgt
158 ( 0x9e  ) INSW ifle
159 ( 0x9f  ) INSW if_icmpeq
160 ( 0xa0  ) INSW if_icmpne
161 ( 0xa1  ) INSW if_icmplt
162 ( 0xa2  ) INSW if_icmpge
163 ( 0xa3  ) INSW if_icmpgt
164 ( 0xa4  ) INSW if_icmple
165 ( 0xa5  ) INSW if_acmpeq
166 ( 0xa6  ) INSW if_acmpne
167 ( 0xa7  ) INSW goto
168 ( 0xa8  ) INSW jsr
169 ( 0xa9  ) INSB ret
170 ( 0xaa  ) INST tableswitch
171 ( 0xab  ) INST lookupswitch
172 ( 0xac  ) INST ireturn
173 ( 0xad  ) INST lreturn
174 ( 0xae  ) INST freturn
175 ( 0xaf  ) INST dreturn
176 ( 0xb0  ) INST areturn
177 ( 0xb1  ) INST return
178 ( 0xb2  ) INSW getstatic
179 ( 0xb3  ) INSW putstatic
180 ( 0xb4  ) INSW getfield
181 ( 0xb5  ) INSW putfield
182 ( 0xb6  ) INSW invokevirtual
183 ( 0xb7  ) INSW invokespecial
184 ( 0xb8  ) INSW invokestatic
185 ( 0xb9  ) INSL invokeinterface
186 ( 0xba  ) INSL invokedynamic
187 ( 0xbb  ) INSW new
188 ( 0xbc  ) INSB newarray
189 ( 0xbd  ) INSW anewarray
190 ( 0xbe  ) INST arraylength
```

```
191 ( 0xbf  ) INST athrow
192 ( 0xc0  ) INSW checkcast
193 ( 0xc1  ) INSW instanceof
194 ( 0xc2  ) INST monitorenter
195 ( 0xc3  ) INST monitorexit
196 ( 0xc4  ) INSL wide
197 ( 0xc5  ) INSL multianewarray
198 ( 0xc6  ) INSW ifnull
199 ( 0xc7  ) INSW ifnonnull
200 ( 0xc8  ) INSL goto_w
201 ( 0xc9  ) INSL jsr_w
202 ( 0xca  ) INSW donext
203 ( 0xcb  ) INSL ldi
204 ( 0xcc  ) INST popr
205 ( 0xcd  ) INST pushr
206 ( 0xce  ) INST dupr
207 ( 0xcf  ) INST ext
208 ( 0xd0  ) INST get
209 ( 0xd1  ) INST put

HEX
: (makeHead)
   20 word                          \ get name of new definition
   lastH @ nameR!                   \ fill link field of last word
   H @ lastH !                      \ save nfa in lastH
   DUP c@ I,                        \ store count
   count FOR AFT
      count I,                      \ fill name field
   THEN NEXT
   DROP
   ;
: makehead
   >IN @ >R                         \ save interpreter pointer
   (makehead)
   R> >IN !                         \ restore interpreter pointer
   ;

: $LIT ( -- )
   22 WORD
   DUP c@ I, ( compile count )
   count FOR AFT
      count I, ( compile characters )
   THEN NEXT    DROP ;

: ldb bipush ;
: ldw sipush ;
: LIT bipush ;

: begin h @ ;
: bz    ifeq ;
: bra   goto ;
: jmp   goto ;

: if      h @ 1+ 0 bz ;
```

```
: ifeqq   h @ 1+ 0 if_icmpeq ;
: ifneg   h @ 1+ 0 iflt ;
: ifgreat h @ 1+ 0 if_icmpgt ;
: ifless  h @ 1+ 0 if_icmplt ;
: skip    h @ 1+ 0 bra ;
: then    begin SWAP ramw! ;
: else    skip  SWAP then ;
: until   bz ;
: while   if SWAP ;
: repeat  bra then ;
: again   bra ;
: aft ( a -- a' a" ) DROP skip begin SWAP ;
: for ( -- a ) pushr begin ;
: next ( a -- ) donext ;
HEAD assembles an header.
```

KERN32i.f

The kernel of JFM is defined in file KERN32i.f. The byte code it refers to are defined in ASM32i.f.

System Variables

Constant and variable store in page 0 memory can be accessed most efficiently by the Java bytecode bipush. A set of system variables are implemented as constants pointing to specific addresses in the variable area, allocated in the beginning of the dictionary.

```
: BASE    40 LIT ;    \ number base
: COMPI   44 LIT ;    \ compile flag
: >IN     48 LIT ;    \ ptr to input char
: HLD     4C LIT ;    \ ptr to output digit
: CONTEXT 50 LIT ;    \ ptr to vocabulary
: LAST    54 LIT ;    \ ptr to last name
: CP      58 LIT ;    \ ptr dictionary top
: DP      5C LIT ;    \ ptr to last dictionary entry
: FENCE   60 LIT ;    \ ptr to boot dictionary
: tmp     64 LIT ;    \ scratch
: ucase   68 LIT ;    \ case insensitive, $FFFFFFDF
: input   6C LIT ;    \ input buffer
: output  70 LIT ;    \ output buffer
```

Assembly Macros

Most of the JFM bytecode will be define as Forth commands, interpreted by the Forth outer interpreter. However, many Java bytecode does not do what Forth commands are required to do exactly. These Forth commands are used extensively in writing the outer interpreter, and must assemble optimized Java bytecode for the best performance. A set of assembly macros are implemented to assemble optimized outer interpreter. Eventually they will be redefined as regular Forth commands which will be interpreted and compiled correctly in the final JFM with less efficiency.

```
: EXIT return ;
: ! ( n a -- ) swap iastore ;
: @ ( a - n ) iaload ;
: C! ( n a -- ) swap bastore ;
: C@ ( a - n ) baload ;
: W! ( n a -- ) swap sastore ;
: W@ ( a - n ) saload ;
: >R ( n ) pushr ;
: R> ( - n ) popr ;
: R@ ( - n ) dupr ;
: DUP ( n - n n ) dup ;
: SWAP ( n1 n2 - n2 n1 )  swap ;
: DROP ( w w  -- )  pop ;
: 2DROP ( w w  -- )  pop2 ;
```

```
: + iadd ;
: - isub ;
: * imul ;
: / idiv ;
: MOD irem ;
: OR ( n n - n )  ior ;
: AND iand ;
: XOR ixor ;
: OVER dup2 pop ;
: NEGATE ( n -- -n )  ineg ;
: 1- ( a -- a )  iconst_m1 iadd ;
: 1+ ( a -- a )  iconst_1 iadd ;
: 2- ( a -- a )  iconst_2 isub ;
: 2+ ( a -- a )  iconst_2 iadd ;
: CELL- ( a -- a )  iconst_4 isub ;
: CELL+ ( a -- a )  iconst_4 iadd ;
: NOT ( w -- w ) iconst_m1 ixor ;
: BL ( -- 32 )  20 LIT ;
: +! ( n a -- )  dup pushr iaload iadd
   popr swap iastore ;
: ROT ( w1 w2 w3 -- w2 w3 w1 )
   pushr swap popr swap ;
: -ROT ( w1 w2 w3 -- w3 w1 w2 )
   dup_x2 pop ;
: 2DUP ( w1 w2 -- w1 w2 w1 w2 )
   dup2 ;
: 2! dup2 ! swap pop iconst_4 iadd ! ;
: 2@ dup @ swap iconst_4 iadd @ swap ;
: COUNT ( b -- b+1 c )
   dup baload swap 1+ swap ;
: 0< ( n - f )
   ifneg iconst_0 else iconst_m1 then ;
: = ( w w -- t )
   ifeqq iconst_0 else iconst_m1 then ;
: > ( n1 n2 - f )
   ifgreat iconst_0 else iconst_m1 then ;
: < ( n1 n2 - f )
   ifless iconst_0 else iconst_m1 then ;
: ?DUP ( w -- w w | 0 )
   dup if dup then ;
: ABS ( n -- +n )
   dup ifneg else ineg then ;
```

```
( ep32s, bytecode & subroutine thread )
HEX
cr .( system variables )
: BASE    40 LIT ;    \ number base
: COMPI   44 LIT ;    \ compile flag
: >IN     48 LIT ;    \ ptr to input char
: HLD     4C LIT ;    \ ptr to output digit
: CONTEXT 50 LIT ;    \ ptr to vocabulary
: LAST    54 LIT ;    \ ptr to last name
: CP      58 LIT ;    \ ptr dictionary top
: DP      5C LIT ;    \ ptr to last dictionary entry
: FENCE   60 LIT ;    \ ptr to boot dictionary
: tmp     64 LIT ;    \ scratch
: ucase   68 LIT ;    \ case insensitive, $FFFFFFDF
: input   6C LIT ;    \ input buffer
: output  70 LIT ;    \ output buffer

cr .( macro words ) cr
: EXIT return ;
: ! ( n a -- ) swap iastore ;
: @ ( a - n ) iaload ;
: C! ( n a -- ) swap bastore ;
: C@ ( a - n ) baload ;
: W! ( n a -- ) swap sastore ;
: W@ ( a - n ) saload ;
: >R ( n ) pushr ;
: R> ( - n ) popr ;
: R@ ( - n ) dupr ;
: DUP ( n - n n ) dup ;
: SWAP ( n1 n2 - n2 n1 )
   swap ;
: DROP ( w w  -- )
   pop ;
: 2DROP ( w w  -- )
   pop2 ;
: + iadd ;
: - isub ;
: * imul ;
: / idiv ;
: MOD irem ;
: OR ( n n - n )
   ior ;
: AND iand ;
: XOR ixor ;
: OVER dup2 pop ;
: NEGATE ( n -- -n )
   ineg ;
: 1- ( a -- a )
   iconst_m1 iadd ;
: 1+ ( a -- a )
   iconst_1 iadd ;
: 2- ( a -- a )
   iconst_2 isub ;
: 2+ ( a -- a )
```

```
   iconst_2 iadd ;
: CELL- ( a -- a )
   iconst_4 isub ;
: CELL+ ( a -- a )
   iconst_4 iadd ;
: NOT ( w -- w ) iconst_m1 ixor ;
: BL ( -- 32 )
   20 LIT ;
: +! ( n a -- )
   dup pushr iaload iadd
   popr swap iastore ;
: ROT ( w1 w2 w3 -- w2 w3 w1 )
   pushr swap popr swap ;
: -ROT ( w1 w2 w3 -- w3 w1 w2 )
   dup_x2 pop ;
: 2DUP ( w1 w2 -- w1 w2 w1 w2 )
   dup2 ;
: 2! dup2 ! swap pop iconst_4 iadd ! ;
: 2@ dup @ swap iconst_4 iadd @ swap ;
: COUNT ( b -- b+1 c )
   dup baload swap 1+ swap ;
: 0< ( n - f )
   ifneg iconst_0 else iconst_m1 then ;
: = ( w w -- t )
   ifeqq iconst_0 else iconst_m1 then ;
: > ( n1 n2 - f )
   ifgreat iconst_0 else iconst_m1 then ;
: < ( n1 n2 - f )
   ifless iconst_0 else iconst_m1 then ;
: ?DUP ( w -- w w | 0 )
   dup if dup then ;
: ABS ( n -- +n )
   dup ifneg else ineg then ;
```

EF32i.f

The dictionary of JFM system contains records of all Forth commands. The low level primitive commands are discussed in the Kernel section. The high level compound commands are discussed here. All compound commands are defined in the file cEFai.f. They are discussed in their loading order. The loading order is very important in the JFM metacompiler, because forward referencing is not allowed. All assembling and compiling processes are accomplished in a single pass.

JFM metacompiler behaves very similar to a regular Forth system. However, to compile primitive commands into a target dictionary, the command CODE was changed to accomplish this goal. To compile high level compound commands to the target dictionary, as we do in this cEFai.f file, a new set of commands :: (colon-colon) and :: (semicolon-semicolon) are used instead of the Forth commands : (colon) and ; (semicolon). Unlike : (colon), :: (colon-colon) does not change to a compiling state, and the metacompiler remains in the interpretive state throughout. New commands defined by the metacompiler would just add new tokens to the target dictionary.

Control structure commands like IF, ELSE, THEN, BEGIN, WHILE, REPEAT, etc, are all redeined in the metacompiler so they can construct control structures properly in the target dictionary. The only exception is the handling of literals. An integer encountered by metacompiler would remain of the data stack. If you intended to compiler it as a literal in the target dictionary, you would have to use the special command LIT. If you are familiar with Forth language, you would notice that the compound commands in cEFai.f read identically like regular Forth code, except that integer literals have to be handled explicitly.

```
: code CODE ;
: ::   CODE ;
: ;;   return ;
:: bye   ext ;;
:: key   get ;;
:: emit  put ;;
```

Common Commands

```
:: max ( n n -- n ) 2DUP      < if SWAP then DROP ;;
:: min ( n n -- n ) 2DUP SWAP < if SWAP then DROP ;;
:: /mod ( n n -- r q )
   2DUP / >R MOD R> ;;
:: */ ( n n n -- q )
   >R * R> / ;;
```

CRR .(Memory access) CRR

```
:: execute ( a ) >R ;;
:: here ( -- a ) CP @ ;;
:: pad ( -- a ) CP @ 50 LIT + ;;
```

```
:: cmove ( b b u -- )
  for aft >R DUP C@ R@ C! 1+ R> 1+
  then next 2DROP ;;
:: fill ( b u c -- )
  SWAP for SWAP aft 2DUP C! 1+ then next 2DROP ;;
```

CRR .(Numeric Output) CRR \ single precision

```
:: digit ( u -- c )
  9 LIT OVER < 7 LIT AND +
  ( CHAR 0 ) 30 LIT + ;;
:: extract ( n base -- n c )
  /mod SWAP digit ;;
:: <# ( -- ) pad HLD ! ;;
:: hold ( c -- ) HLD @ 1- DUP HLD ! C! ;;
:: # ( u -- u ) BASE @ extract hold ;;
:: #s ( u -- 0 ) begin # DUP while repeat ;;
CRR
:: sign ( n -- ) 0< if ( CHAR - ) 2D LIT hold then ;;
:: #> ( w -- b u ) DROP HLD @ pad OVER - ;;
:: str ( n -- b u ) DUP >R ABS <# #s R> sign #> ;;
:: hex ( -- ) 10 LIT BASE ! ;;
:: decimal ( -- ) 0A LIT BASE ! ;;
```

>CHAR is very important in converting a non-printable character to a harmless 'underscore' character (ASCII 95). As eForth is designed to communicate with you through a serial I/O device, it is important that eForth will not emit control characters to the host and causes unexpected behavior on the host computer. >CHAR thus filters the characters before they are sent out by TYPE.

```
:: >CHAR ( c -- c )
  $7F LIT AND DUP $7F LIT BL WITHIN
  IF DROP ( CHAR _ ) $5F LIT THEN ;;
```

ALIGNED changes the address to the next cell boundary so that it can be used to address 32 bit word in memory.

```
:: ALIGNED ( b -- a ) 3 LIT + FFFFFFFC LIT AND ;;
```

HERE returns the address of the first free location above the code dictionary, where new commands are compiled.

```
:: HERE ( -- a ) CP @ ;;
```

PAD returns the address of the text buffer where numbers are constructed and text strings are stored temporarily.

```
:: PAD ( -- a ) HERE 50 LIT + ;;
```

TIB returns the terminal input buffer where input text string is held.

```
:: TIB ( -- a ) 'TIB @ ;;
```

@EXECUTE is a special command supporting the vectored execution commands in eForth. It fetches the code field address of a token and executes the token.

```
:: @EXECUTE ( a -- ) @ ?DUP IF EXECUTE THEN ;;
```

CMOVE copies a memory array from one location to another. It copies one byte at a time.

```
:: CMOVE ( b b u -- )
  FOR AFT OVER c@ OVER c! >R 1+ R> 1+ THEN NEXT 2DROP ;;
```

MOVE copies a memory array from one location to another. It copies one word at a time.

```
:: MOVE ( b b u -- )
  CELL/ FOR AFT OVER @ OVER ! >R CELL+ R> CELL+ THEN NEXT 2DROP ;;
```

FILL fills a memory array with the same byte.

```
:: FILL ( b u c -- )
  SWAP FOR SWAP CRR .( Basic I/O ) CRR


:: space ( -- ) BL emit ;;
:: spaces ( +n -- ) for aft space then next ;;
:: >char ( c -- c )
  $7E LIT min BL max ;;
:: type ( b u -- )
  for aft COUNT ( >char ) emit
  then next DROP ;;
:: cr ( -- ) ( =Cr )
  0A LIT 0D LIT emit emit ;;
:: do$ ( -- a , get prior frame )


CRR
:: $"| ( -- a ) do$ ;;
:: ."| ( -- ) do$ COUNT type ;;
::  .r ( n +n -- )
  >R str R> OVER - spaces type ;;
::  . ( n -- )
  str space type ;;
:: ? ( a -- ) @ . ;;
  R> R> DUP COUNT +  >R SWAP >R ;; AFT 2DUP c! 1+ THEN NEXT 2DROP ;;
```

CRR .(Numeric Input) CRR \ single precision

```
:: digit? ( c base -- u t | x 0 )
  >R ( c ) DUP 40 LIT >
  if 5F LIT AND 37 LIT - ( above @ )
  else DUP 39 LIT >
    if DROP 7F LIT ( above 9 )
    else 30 LIT - DUP ( 0-9 )
      0< if ( below 0 ) DROP 7F LIT then
    then
  then DUP R> ( u u base ) < ( u t | x 0 )
  ;;
```

```
:: number? ( a -- n T | a F )
  DUP >R COUNT >R ( a+1 )
  COUNT ( a+2 c - ) 2D LIT = ( a+2 f )
  DUP tmp !
  if R> 1- ( a+2 n-1 )
  else 1- R> ( a+1 n )
  then tmp @ >R iconst_0 tmp !
  for aft ( a' )
    COUNT ( a'+1 c ) BASE @ digit? ( a'+1 b  f )
    if tmp @ BASE @ * + tmp !
    else 2DROP R> R> 2DROP R> iconst_0 ( a 0 ) EXIT
    then
  then next
  DROP tmp @ ( u ) R> if NEGATE then
  iconst_m1 R> DROP ;; ( u t )
```

CRR .(Parsing) CRR

```
:: parse ( c a -- a-1 )
  DUP tmp ! >IN ! ( c )
  begin key DUP emit DUP >IN @ C! $20 LIT > until
  begin iconst_1 >IN +! ( c )
    key DUP emit 2DUP XOR ( c k f1 )
    OVER $1F LIT > ( c k f1 f2 ) AND ( c k f )
  while >IN @ C! ( c )
  repeat ( c k )
  2DROP tmp @ >IN @ OVER - ( a n )
  SWAP 1- SWAP ( a-1 n )
  OVER C! ( a-1 )
  ;;
:: token ( -- a , parser buffer )
  BL CP @ iconst_3 + parse  ;;
:: word ( c -- a , word buffer )
  CP @ 1+ parse ;;
```

CRR .(Dictionary Search) CRR

```
:: name> ( na -- ca ) COUNT 1F LIT AND +  ;;
:: same? ( a na -- a na diff )
  OVER W@ OVER W@ ( a na ca cna )
  $1FFF ldi AND XOR ucase @ AND ?DUP if EXIT ( a na diff ) then
  OVER C@ 1- >R ( a na )
  OVER 2+ OVER 2+ R> ( a na a+1 na+1 length )
  for aft OVER R@ + C@ ( a na a+i na+i ca )
    OVER R@ + C@ ( a na a+i na+i ca cna )
    XOR ucase @ AND ( a na a+i na+i diff )
    if R> 2DROP ( a na a+i ) EXIT then
```

```
  then next ( a na a+i na+i )
  2DROP iconst_0  ;; ( a na 0 )

:: name? ( a -- cfa nfa | a 0 )
  CONTEXT ( a va )
  begin W@ DUP  ( a na na )
  while ( a na )
    same? ( a na f )
    if 2- ( a la ) DUP tmp ! ( save for see )
    else SWAP DROP DUP name> SWAP EXIT ( ca na )
    then
  repeat ;; ( a 0 --, dictionary start )
```

CRR .(compiler) CRR

```
:: [ ( -- )
  iconst_0 COMPI ! ;; IMMEDIATE
:: ] ( -- )
  iconst_m1 COMPI ! ;;
:: , ( n -- ) here DUP CELL+ CP ! ! CP @ DP ! ;;
:: w, ( w -- ) here DUP 2+ CP ! W! CP @ DP ! ;;
:: c, ( c -- ) here DUP 1+ CP ! C! CP @ DP ! ;;
:: allot ( n -- )
   for aft iconst_0 c, then next ;;
:: compil ( w -- ) B6 LIT c, w, ;;
:: literal ( n )
   DUP 0< if CB LIT c, , EXIT then
   DUP $100 ldi < if 10 LIT c, c, EXIT then
   DUP $10000 ldi <
   if 11 LIT c, w,
   else CB LIT c, ,
   then ;;
```

CRR (outer interpreter)

```
:: ok ( -- )
  COMPI @ if else
    cr >R >R >R DUP .
    R> DUP . R> DUP . R> DUP .
    ."| $LIT >ok "
  then ;;
:: quit ( -- )
  [ ( outer interpret )
  begin
    token ( a )
    name? ( ca na | a 0 )
    ?DUP ( ca na na | a 0 )
    if ( ca na )
      C@ $80 LIT AND ( ca immd )
```

```
      if ( ca ) execute
      else
        COMPI @ if compil else execute then
      then
    else ( a )
      number? ( n f | a 0 )
      if ( n ) COMPI @ if literal then
      else ( a )
        DP @ CP ! ( clean dictionary )
        COUNT type $3F LIT emit cr [
      then
    then
    COMPI @ if else ok then
  again
:: abort"| ( f -- )
  if do$ COUNT type quit then do$ DROP ;;
:: error ( a -- )
  space COUNT type $3F LIT emit cr quit
```

CRR .(colon compiler) CRR

```
:: compile ( -- )
  R> 1+ DUP W@ compil
  2+ >R ;;
:: ?unique ( a -- a )
  DUP name?
  if COUNT type ."| $LIT reDef "
  then DROP ;;
:: $,n ( a -- )
  DUP @
  if ?unique
    ( na) CP @ DP !
    ( na) DUP name> CP !
    ( na) DUP LAST ! \ for overt
    ( na) 2-
    ( la) CONTEXT W@ SWAP W! EXIT
  then CRR .( Tools ) CRR
:: ' ( -- ca )
  token name? if EXIT then
  error
:: dm+ ( b u -- b+u )
  OVER 6 LIT .r space
  for aft DUP C@ 3 LIT .r 1+
  then next ;;
:: dump ( b u -- )
  hex  10 LIT /
  for aft cr 10 LIT dm+ space
```

```
    DUP 10 LIT - 10 LIT type
  then next DROP ;;

CRR
:: >name ( ca -- na | F )
  CONTEXT ( ca la )
  begin W@ DUP ( ca na na )
  while 2DUP name> ( ca na ca ca ) XOR
    if 2- ( ca la )
    else SWAP DROP EXIT ( na )
    then
  repeat SWAP DROP ( na=0 ) ;;
:: id ( a -- )
  COUNT $01F LIT AND type space ;;

CRR
:: see ( -- ; <string> )
  cr ' ( ca --, tmp has next la )
  begin ( ca )
    COUNT DUP $B6 LIT XOR ( ca+1 b f )
    if . space
    else DROP COUNT >R ( ca+2 )
      COUNT $100 ldi * R> + ( ca+3 w ) >name
      ?DUP if id else $5F LIT emit space then
      1+ ( ca+4 )
    then
    DUP tmp @ > ( ca+4 )
  until DROP ;;
:: words ( -- )
  cr CONTEXT
  begin W@ ?DUP
  while DUP id 2-
  repeat cr ;;
:: case if $FFFFFFDF ldi else $FFFFFFFF ldi then ucase ! ;;error
:: overt ( -- ) LAST @ CONTEXT W! ;;
:: ; ( -- )
  B1 LIT c, [ overt ;; IMMEDIATE
:: : ( -- ; <string> )
  token $,n ] ;;
```

Numeric Output

`DIGIT` converts an integer to an ASCII digit.

```
:: DIGIT ( u -- c )
  9 LIT OVER < 7 LIT AND +
  ( CHAR 0 ) 30 LIT + ;;
```

EXTRACT extracts the least significant digit from a number n. n is divided by the radix in BASE and returned on the stack.

```
:: EXTRACT ( n base -- n c )
  0 LIT SWAP UM/MOD SWAP DIGIT ;;
```

<# initiates the output number conversion process by storing PAD buffer address into variable HLD, which points to the location next numeric digit will be stored.

```
:: <# ( -- ) PAD HLD ! ;;
```

HOLD appends an ASCII character whose code is on the top of the parameter stack, to the numeric output string at HLD. HLD is decremented to receive the next digit.

```
:: HOLD ( c -- ) HLD @ 1- DUP HLD ! C!  ;;
```

(dig) extracts one digit from integer on the top of the parameter stack, according to radix in BASE, and add it to output numeric string.

```
:: # ( u -- u ) BASE @ EXTRACT HOLD ;;
```

#S (digs) extracts all digits to output string until the integer on the top of the parameter stack is divided down to 0.

```
:: #S ( u -- 0 ) BEGIN # DUP WHILE REPEAT ;;
```

SIGN inserts a - sign into the numeric output string if the integer on the top of the parameter stack is negative.

```
:: SIGN ( n -- ) 0< IF ( CHAR - ) 2D LIT HOLD THEN ;;
```

#> terminates the numeric conversion and pushes the address and length of output numeric string on the parameter stack.

```
:: #> ( w -- b u ) DROP HLD @ PAD OVER - ;;
```

str converts a signed integer on the top of data stack to a numeric output string.

```
:: str ( n -- b u ) DUP >R ABS <# #S R> SIGN #> ;;
```

HEX sets numeric conversion radix in BASE to 16 for hexadecimal conversions.

```
:: HEX ( -- ) 10 LIT BASE ! ;;
```

DECIMAL sets numeric conversion radix in BASE to 10 for decimal conversions.

```
:: DECIMAL ( -- ) 0A LIT BASE ! ;;
```

Numeric Input

wupper converts 4 bytes in a word to upper case characters.

```
:: wupper ( w -- w' ) 5F5F5F5F LIT AND ;;
```

>upper converts a character to upper case.

```
:: >upper ( c -- UC )
```

```
  dup 61 LIT 7B LIT WITHIN IF 5F LIT AND THEN ;;
```

DIGIT? converts a digit to its numeric value according to the current base, and NUMBER? converts a number string to a single integer.

```
:: DIGIT? ( c base -- u t )
  >R ( CHAR 0 ) >upper 30 LIT - 9 LIT OVER <
  IF 7 LIT - DUP 0A LIT  < OR THEN DUP R> U< ;;
```

NUMBER? converts a string of digits to a single integer. If the first character is a $ sign, the number is assumed to be in hexadecimal. Otherwise, the number will be converted using the radix value stored in BASE. For negative numbers, the first character should be a - sign. No other characters are allowed in the string. If a non-digit character is encountered, the address of the string and a false flag are returned. Successful conversion returns the integer value and a true flag. If the number is larger than 2**n, where n is the bit width of a single integer, only the modulus to 2**n will be kept.

```
:: NUMBER? ( a -- n T | a F )
  BASE @ >R  0 LIT OVER COUNT ( a 0 b n)
  OVER c@ ( CHAR $ ) 24 LIT =
  IF HEX SWAP 1+ SWAP 1- THEN ( a 0 b' n')
  OVER c@ ( CHAR - ) 2D LIT = >R ( a 0 b n)
  SWAP R@ - SWAP R@ + ( a 0 b" n") ?DUP
  IF 1- ( a 0 b n)
    FOR DUP >R c@ BASE @ DIGIT?
      WHILE SWAP BASE @ * +  R> 1+
    NEXT DROP R@ ( b ?sign) IF NEGATE THEN SWAP
      ELSE R> R> ( b index) 2DROP ( digit number) 2DROP 0 LIT
      THEN DUP
  THEN R> ( n ?sign) 2DROP R> BASE ! ;;
```

Character Output

SPACE outputs a blank space character.

```
:: SPACE ( -- ) BL EMIT ;;
```

CHARS outputs n characters c.

```
:: CHARS ( +n c -- )
  SWAP 0 LIT MAX
  FOR AFT DUP EMIT THEN NEXT DROP ;;
```

SPACES outputs n blank space characters.

```
:: SPACES ( +n -- ) BL CHARS ;;
```

TYPE outputs n characters from a string in memory. Non ASCII characters are replaced by a underscore character.

```
:: TYPE ( B U -- )
  FOR AFT DUP C@ >CHAR EMIT 1+ THEN NEXT DROP ;;
```

CR outputs a carriage-return and a line-feed. Prior output characters are accumulated in a UDP packet buffer. This packet is sent out by sendPacket.

```
:: CR ( -- ) ( =CR )
  0A LIT 0D LIT EMIT EMIT sendPacket ;;
```

do$ retrieves the address of a string stored as the second item on the return stack. do$ is a bit difficult to understand, because the starting address of the following string is the second item on the return stack. This address is pushed on the data stack so that the string can be accessed. This address must be changed so that the address interpreter will return to the token right after the compiled string. This address will allow the address interpreter to skip over the string literal and continue to execute the token list as intended. Both $"| and ."| use the command do$,

```
:: do$ ( -- $adr )
  R> R@ R> COUNT + ALIGNED >R SWAP >R ;;
```

$"| push the address of the following string on stack. Other commands can use this address to access data stored in this string. The string is a counted string. Its first byte is a byte count.

```
:: $"| ( -- a ) do$ ;;
```

."| displays the following string on stack. This is a very convenient way to send helping messages to you at run time.

```
:: ."| ( -- ) do$ COUNT TYPE ;;
```

.R displays a signed integer n , the second item on the parameter stack, right-justified in a field of +n characters. +n is on the top of the parameter stack.

```
::  .R ( n +n -- )
  >R str R> OVER - SPACES TYPE ;;
```

U.R displays an unsigned integer n right-justified in a field of +n characters.

```
:: U.R ( u +n -- )
  >R <# #S #> R> OVER - SPACES TYPE ;;
```

U. displays an unsigned integer u in free format, followed by a space.

```
:: U. ( u -- ) <# #S #> SPACE TYPE ;;
```

. (dot) displays a signed integer n in free format, followed by a space.

```
::  . ( n -- )
  BASE @ 0A LIT  XOR
  IF U. EXIT THEN str SPACE TYPE ;;
```

? displays signed integer stored in memory a on the top of the parameter stack, in free format followed by a space.

```
:: ? ( a -- ) @ . ;;
```

Parser

(parse) (b1 u1 c --b2 u2 n) From the source string starting at b1 and of u1 characters long, parse out the first word delimited by character c. Return the address b2 and length u2 of the word just parsed out and the difference n between b1 and b2. Leading delimiters are skipped over. (parse) is used by PARSE.

```
:: (parse) ( b u c -- b u delta ; <string> )
  tmp c! OVER >R DUP \ b u u
  IF 1- tmp c@ BL =
    IF              \ b u' \ 'skip'
```

```
      FOR BL OVER c@ - 0< NOT
        WHILE 1+
      NEXT ( b) R> DROP 0 LIT DUP EXIT \ all delim
        THEN  R>
    THEN OVER SWAP  \ b' b' u' \ 'scan'
    FOR tmp c@ OVER c@ -  tmp c@ BL =
      IF 0< THEN WHILE 1+
    NEXT DUP >R
      ELSE R> DROP DUP 1+ >R
      THEN OVER -  R>  R> - EXIT
  THEN ( b u) OVER R> - ;;
```

PACK$ copies a source string (b u) to target address at a. The target string is null filled to the cell boundary. The target address a is returned.

```
:: PACK$ ( b u a -- a ) \ always word-aligned
  DUP >R
  2DUP + $FFFFFFFC LIT AND 0 LIT SWAP ! \ LAST WORD FILL 0 1ST
  2DUP C! 1+ SWAP CMOVE  R> ;;
```

PARSE scans the source string in the terminal input buffer from where >IN points to till the end of the buffer, for a word delimited by character c. It returns the address and length of the word parsed out. PARSE calls (parse) to do the dirty work.

```
:: PARSE ( c -- b u ; <string> )
  >R  TIB >IN @ +
  #TIB @ >IN @ -
  R> (parse) >IN +! ;;
```

TOKEN parses the next word from the input buffer and copy the counted string to the top of the name dictionary. Return the address of this counted string.

```
:: TOKEN ( -- a ;; <string> )
  BL PARSE $1F LIT MIN
  HERE CELL+         \ S D N
  PACK$  ;;
```

WORD parses out the next word delimited by the ASCII character c. Copy the word to the top of the code dictionary and return the address of this counted string.

```
:: WORD ( c -- a ; <string> )
  PARSE HERE CELL+ PACK$ ;; \ BM+
```

Dictionary Search

NAME> (nfa – cfa) Return a code field address from the name field address of a command.

```
:: NAME> ( a -- xt ) COUNT 1F LIT AND + ALIGNED ;;
```

SAME? (a1 a2 n – a1 a2 f) Compare n/4 words in strings at a1 and a2. If the strings are the same, return a 0. If string at a1 is higher than that at a2, return a positive number; otherwise, return a negative number. FIND compares the 1st word input string and a name. If these two words are the same, SAME? is called to compare the rest of two strings

```
:: SAME? ( a a u -- a a f \ -0+ )
 $1F LIT AND CELL/
```

```
 FOR AFT OVER R@ 4 LIT * + @ wupper
   OVER R@ 4 LIT * + @ wupper
   - ?DUP IF R> DROP EXIT THEN
 THEN NEXT
 0 LIT ;;
```

find (a va --cfa nfa, a F) searches the dictionary for a command. A counted string at a is the name of a token to be looked up in the dictionary. The last name field address of the dictionary is stored in location va. If the string is found, both the code field address and the name field address are returned. If the string is not the name a token, the string address and a false flag are returned.

```
:: find ( a va -- xt na | a 0 )
  SWAP         \ va a
  DUP @ tmp !  \ va a  \ get cell count
  DUP @ >R     \ va a  \ #XOR --- count and 1st 3 char
  cell+ SWAP      \ a' va  a'=a(#XOR)+4
  BEGIN @ DUP  \ a' na na
    IF DUP @ $FFFFFF3F LIT AND wupper
      R@ wupper XOR \ ignore lexicon bits
      IF cell+ -1 LIT
      ELSE cell+ tmp @ SAME?
      THEN
    ELSE R> DROP SWAP cell- SWAP EXIT \ a 0
    THEN
  WHILE cell- cell-  \ a' la
  REPEAT R> DROP SWAP DROP
  cell- DUP NAME> SWAP ;;
:: NAME? ( a -- cfa na | a 0 )
  CONTEXT find ;;
```

Text Interpreter

EXPECT (b u1 --) accepts u1 characters to b. Number of characters accepted is stored in SPAN.

```
:: EXPECT ( b u -- ) accept SPAN ! DROP ;;
```

QUERY is the command which accepts text input, up to 80 characters, from an input device and copies the text characters to the terminal input buffer. It also prepares the terminal input buffer for parsing by setting #TIB to the received character count and clearing >IN.

```
:: QUERY ( -- )
  TIB 50 LIT ACCEPT #TIB !
  DROP 0 LIT >IN ! ;;
```

ABORT resets system and re-enters into the text interpreter loop QUIT. It actually executes QUIT stored in ‘ABORT. This avoids forward-referencing to QUIT, as QUIT is yet to be defined.

```
:: ABORT ( -- ) 'ABORT @EXECUTE ;;
```

abort"| (f --) A runtime string command compiled in front of a string of error message. If flag f is true, display the following string and jump to ABORT. If flag f is false, ignore the following string and continue executing tokens after the error message.

```
:: abort" ( f -- )
  IF do$ COUNT TYPE ABORT THEN do$ DROP ;;
```

ERROR displays an error message at a with a ? mark, and ABORT.

```
:: ERROR ( a -- )
  space count type $3F LIT EMIT
  $1B LIT ( ESC) EMIT
  CR ABORT
```

$INTERPRET executes a command whose string address is on the stack. If the string is not a command, convert it to a number. If it is not a number, ABORT.

```
:: $INTERPRET ( a -- )
  NAME? ?DUP
  IF C@ $40 LIT AND
    abort" $LIT  compile only" ( ?even)  EXECUTE  EXIT
  THEN
  NUMBER? IF EXIT ELSE ERROR THEN ;;
```

[(left-paren) activates the text interpreter by storing the execution address of $INTERPRET into the variable 'EVAL, which is executed in EVAL while the text interpreter is in the interpretive mode.

```
:: [ ( -- )  DOLIT $INTERPRET 'EVAL ! ;; IMMEDIATE
```

.OK used to be a command which displays the familiar 'ok' prompt after executing to the end of a line. In JFM_44, it displays the top 4 elements on data stack so you can see what is happening on the stack. It is more informative than the plain ‘ok’, which only give you a warm and fuzzy feeling about the system. When text interpreter is in compiling mode, the display is suppressed.

```
:: .OK ( -- ) CR
  DOLIT $INTERPRET 'EVAL @ =
  IF >R >R >R DUP . R> DUP . R> DUP . R> DUP . ."| $LIT  fg>"
  THEN ;;
```

EVAL has a loop which parses tokens from the input stream and invokes whatever is in 'EVAL to process that token, either execute it with $INTERPRET or compile it with $COMPILE. It exits the loop when the input stream is exhausted.

```
:: EVAL ( -- )
  BEGIN TOKEN DUP @
  WHILE 'EVAL  @EXECUTE \ ?STACK
  REPEAT DROP .OK ;;
```

QUIT is the operating system, or a shell, of the eForth system. It is an infinite loop eForth will not leave. It uses QUERY to accept a line of text from the terminal and then let EVAL parse out the tokens and execute them. After a line is processed, it displays the top of data stack and wait for the next line of text. When an error occurred during execution, it displays the command which caused the error with an error message. After the error is reported, it re-initializes the

system by jumping to ABORT. Because the behavior of EVAL can be changed by storing either $INTERPRET or $COMPILE into 'EVAL, QUIT exhibits the dual nature of a text interpreter and a compiler.

```
:: QUIT ( -- ) [ BEGIN QUERY EVAL AGAIN
```

Command Compiler

, (comma) adds the execution address of a token on the top of the data stack to the code dictionary, and thus compiles a token to the growing token list of the command currently under construction.

```
:: , ( w -- )  HERE DUP CELL+ CP ! ! ;;
```

LITERAL compiles an integer literal to the current compound command under construction. The integer literal is taken from the data stack, and is preceded by the token DOLIT. When this compound command is executed, DOLIT will extract the integer from the token list and push it back on the data stack. LITERAL compiles an address literal if the compiled integer happens to be an execution address of a token. The address will be pushed on the data stack at the run time by DOLIT.

```
:: LITERAL ( n -- ) DOLIT DOLIT , , ;; IMMEDIATE
```

ALLOT allocates n bytes of memory on the top of the dictionary. Once allocated, the compiler will not touch the memory locations. It is possible to allocate and initialize this array using the command', (comma)'.

```
:: ALLOT ( n -- ) ALIGNED CP +! ;;
:: $," ( -- ) ( CHAR " ) 22 LIT WORD COUNT + ALIGNED CP ! ;;
```

?UNIQUE is used to display a warning message to show that the name of a new command already existing in dictionary. eForth does not mind your reusing the same name for different commands. However, giving many commands the same name is a potential cause of problems in maintaining software projects. It is to be avoided if possible and ?UNIQUE reminds you of it.

```
:: ?UNIQUE ( a -- a )
  DUP NAME?
  ?DUP IF COUNT 1F LIT AND SPACE TYPE ."| $LIT  reDef "
  THEN DROP ;;
```

$,n builds a new name field in dictionary using the name already moved to the top of dictionary by PACK$. It pads the link field with the address stored in LAST. A new token can now be built in the code dictionary.

```
:: $,n ( a -- )
  DUP @ IF ?UNIQUE
    ( na ) DUP NAME> CP !
    ( na ) DUP LAST ! \ for OVERT
    ( na ) CELL-
    ( la ) CONTEXT @ SWAP ! EXIT
  THEN ERROR
```

' (tick) searches the next word in the input stream for a token in the dictionary. It returns the code field address of the token if successful. Otherwise, it displays an error message.

```
:: ' ( -- xt )
  TOKEN NAME? IF EXIT THEN
  ERROR
```

[COMPILE] acts similarly, except that it compiles the next command immediately. It causes the following command to be compiled, even if the following command is usually an immediate command which would otherwise be executed.

```
:: [COMPILE] ( -- ; <string> )
  '  , ;; IMMEDIATE
```

COMPILE is used in a compound command. It causes the next token after COMPILE to be added to the top of the code dictionary. It therefore forces the compilation of a token at the run time.

```
:: COMPILE ( -- ) R> DUP @ , CELL+ >R ;;
```

$COMPILE builds the body of a new compound command. A complete compound command also requires a header in the name dictionary, and its code field must start with a call, byte code. These extra works are performed by : (colon). Compound commands are the most prevailing type of commands in eForth. In addition, eForth has a few other defining commands which create other types of new commands in the dictionary.

```
:: $COMPILE ( a -- )
  NAME? ?DUP
  IF @ $80 LIT AND
    IF EXECUTE
    ELSE ,
    THEN EXIT
  THEN
  NUMBER?
  IF LITERAL EXIT
  THEN ERROR
```

OVERT links a new command to the dictionary and thus makes it available for dictionary searches.

```
:: OVERT ( -- ) LAST @ CONTEXT ! ;;
```

] (right paren) turns the interpreter to a compiler.

```
:: ] ( -- ) DOLIT $COMPILE 'EVAL ! ;;
```

: (colon) creates a new header and start a new compound command. It takes the following string in the input stream to be the name of the new compound command, by building a new header with this name in the name dictionary. It then compiles a call, byte code at the beginning of the code field in the code dictionary. Now, the code dictionary is ready to accept a token list.] (right paren) is now invoked to turn the text interpreter into a compiler, which will compile the following words in the input stream to a token list in the code dictionary. The new compound command is terminated by ;, which compiles an EXIT to terminate the token list, and executes [(left paren) to turn the compiler back to text interpreter.

```
:: : ( -- ; <string> ) TOKEN $,n ] 6 LIT , ;;
```

; (semi-colon) terminates a compound command. It compiles an EXIT to the end of the token list, links this new command to the dictionary, and then reactivates the interpreter.

```
:: ; ( -- ) DOLIT EXIT , [ OVERT  ;; IMMEDIATE
```

Debugging Tools

dm+ dumps u bytes starting at address b to the terminal. It dumps 8 words. A line begins with the address of the first byte, followed by 8 words shown in hex, and the same data shown in ASCII. Non-printable characters by replaced by underscores. A new address b+u is returned to dump the next line.

```
:: dm+ ( b u -- b )
  OVER 6 LIT U.R
  FOR AFT DUP @ 9 LIT U.R CELL+
  THEN NEXT ;;
```

DUMP dumps u bytes starting at address b to the terminal. It dumps 8 words to a line. A line begins with the address of the first byte, followed by 8 words shown in hex. At the end of a line are the 32 bytes shown in ASCII code.

```
:: DUMP ( b u -- )
  BASE @ >R HEX  1F LIT + 20 LIT /
  FOR AFT CR 8 LIT 2DUP dm+
  >R SPACE CELLS TYPE R>
  THEN NEXT DROP R> BASE ! ;;
```

>NAME finds the name field address of a token from its code field address. If the token does not exist in the dictionary, it returns a false flag. >NAME is the mirror image of the command NAME>, which returns the code field address of a token from its name field address. Since the code field is right after the name field, whose length is stored in the lexicon byte, NAME> is trivial. >NAME is more complicated because we have to search the dictionary to acertain the name field address.

```
:: >NAME ( xt -- na | F )
  CONTEXT
  BEGIN @ DUP
  WHILE 2DUP NAME> XOR
    IF 1-
    ELSE SWAP DROP EXIT
    THEN
  REPEAT SWAP DROP ;;
```

.ID displays the name of a token, given its name field address. It also replaces non-printable characters in a name by under-scores.

```
:: .ID ( a -- )
  COUNT $01F LIT AND TYPE SPACE ;;
```

WORDS displays all the names in the dictionary. The order of commands is reversed from the compiled order. The last defined command is shown first.

```
:: WORDS ( -- )
```

```
  CR CONTEXT
  0 LIT TMP !
  BEGIN @ ?DUP
  WHILE DUP SPACE  .ID CELL-
    TMP @ 10 LIT <
    IF 1 LIT TMP +!
    ELSE CR 0 LIT TMP ! THEN
  REPEAT ;;
```

FORGET searches the dictionary for a name following it. If it is a valid command, trim dictionary below this command. Display an error message if it is not a valid command.

```
:: FORGET ( -- )
  TOKEN NAME? ?DUP
  IF CELL- DUP CP !
     @ DUP CONTEXT ! LAST !
     DROP EXIT
  THEN ERROR
```

COLD is a high level word executed upon power-up. It sends out sign-on message, and then falls into the text interpreter loop through QUIT.

```
:: COLD ( -- )
   CR ."| $LIT JFM V4.3, 2017 " CR
   QUIT ;;
```

Control Structures

THEN terminates a conditional branch structure. It uses the address of next token to resolve the address literal at A left by IF or ELSE.

```
:: THEN ( A -- ) HERE SWAP ! ;; IMMEDIATE
```

FOR starts a FOR-NEXT loop structure in a colon definition. It compiles >R, which pushes a loop count on return stack. It also leaves the address of next token on data stack, so that NEXT will compile a DONEXT address literal with the correct branch address.

```
:: FOR ( -- a ) COMPILE >R HERE ;; IMMEDIATE
```

BEGIN starts an infinite or indefinite loop structure. It does not compile anything, but leave the current token address on data stack to resolve address literals compiled later.

```
:: BEGIN ( -- a ) HERE ;; IMMEDIATE
```

NEXT Terminate a FOR-NEXT loop structure, by compiling a DONEXT address literal, branch back to the address A on data stack.

```
:: NEXT ( a -- )  COMPILE DONEXT , ;; IMMEDIATE
```

UNTIL terminate a BEGIN-UNTIL indefinite loop structure. It compiles a QBRANCH address literal using the address on data stack.

```
:: UNTIL ( a -- ) COMPILE QBRANCH , ;; IMMEDIATE
```

AGAIN terminate a BEGIN-AGAIN infinite loop structure. . It compiles a BRANCH address literal using the address on data stack.

```
:: AGAIN ( a -- ) COMPILE BRANCH , ;; IMMEDIATE
```

IF starts a conditional branch structure. It compiles a QBRANCH address literal, with a 0 in the address field. It leaves the address of this address field on data stack. This address will later be resolved by ELSE or THEN in closing the true clause in the branch structure.

```
:: IF ( -- A )   COMPILE QBRANCH HERE 0 LIT , ;; IMMEDIATE
```

AHEAD starts a forward branch structure. It compiles a BRANCH address literal, with a 0 in the address field. It leaves the address of this address field on data stack. This address will later be resolved when the branch structure is closed.

```
:: AHEAD ( -- A ) COMPILE BRANCH HERE 0 LIT , ;; IMMEDIATE
```

REPEAT terminates a BEGIN-WHILE-REPEAT indefinite loop structure. It compiles a BRANCH address literal with address a left by BEGIN, and uses the address of next token to resolve the address literal at A.

```
:: REPEAT ( A a -- ) AGAIN THEN ;; IMMEDIATE
```

AFT jumps to THEN in a FOR-AFT-THEN-NEXT loop the first time through. It compiles a BRANCH address literal and leaves its address field on stack. This address will be resolved by THEN. It also replaces address A left by FOR by the address of next token so that NEXT will compile a DONEXT address literal to jump back here at run time.

```
:: AFT ( a -- a A ) DROP AHEAD HERE SWAP ;; IMMEDIATE
```

ELSE (A--A) starts the false clause in an IF-ELSE-THEN structure. It compiles a BRANCH address literal. It uses the current token address to resolve the branch address in A, and replace A with the address of its address literal.

```
:: ELSE ( A -- A )  AHEAD SWAP THEN ;; IMMEDIATE
```

WHILE (a--Aa) compiles a QBRANCH address literal in a BEGIN-WHILE-REPEAT loop. The address A of this address literal is swapped with address a left by BEGIN, so that REPEAT will resolve all loose ends and build the loop structure correctly.

```
:: WHILE ( a -- A a )    IF SWAP ;; IMMEDIATE
```

String Literals

ABORT" compiles an error message. This error message is display if top item on the stack is non-zero. The rest of the commands in the command is skipped and eForth resets to ABORT. If top of stack is 0, ABORT″ skips over the error message and continue executing the following token list.

```
:: ABORT" ( -- ; <string> ) DOLIT abort" HERE ! $," ;; IMMEDIATE
```

$" compiles a character string. When it is executed, only the address of the string is left on the data stack. You will use this address to access the string and individual characters in the string as a string array.

```
:: $"     ( -- ; <string> ) DOLIT $"|    HERE ! $," ;; IMMEDIATE
```

." (dot-quot) compiles a character string which will be displayed when the command containing it is executed in the runtime. This is the best way to present messages to the user.

```
:: ."     ( -- ; <string> ) DOLIT ."|    HERE ! $," ;; IMMEDIATE
```

Defining Commands

CODE creates a command header, ready to accept byte code for a new primitive command. Without a byte code assembler, you can use the command , (comma) to add words with byte code in them.

```
:: CODE ( -- ; <string> ) TOKEN $,n OVERT align ;;
```

CREATE creates a new array without allocating memory. Memory is allocated using ALLOT.

```
:: CREATE ( -- ; <string> ) CODE $203D LIT , ;;
```

VARIABLE creates a new variable, initialized to 0.

```
:: VARIABLE ( -- ; <string> ) CREATE 0 LIT , ;;
```

CONSTANT creates a new constant, initialized to the value on top of stack.

```
:: CONSTANT CODE $2004  LIT , , ;;
```

Immediate Commands

.((dot-paren) types the following string till the next). It is used to output text to the terminal.

```
(makeHead) .( ( -- ) call, aanew 29 LIT PARSE TYPE ;; IMMEDIATE
```

\ (back-slash) ignores all characters till end of input buffer. It is used to insert comment lines in text.

```
(makeHead) \ ( -- )  call, aanew $A LIT WORD DROP  ;; IMMEDIATE
```

((paren) ignores the following string till the next). It is used to place comments in source text.

```
(makeHead) ( call, aanew 29 LIT PARSE 2DROP ;;        IMMEDIATE
```

COMPILE-ONLY sets the compile-only lexicon bit in the name field of the new command just compiled. When the interpreter encounters a command with this bit set, it will not execute this command, but spit out an error message. This bit prevents structure commands to be executed accidentally outside of a compound command.

```
 (makeHead) COMPILE-ONLY call, aanew $40 LIT LAST @ +! ;;
```

IMMEDIATE sets the immediate lexicon bit in the name field of the new command just compiled. When the compiler encounters a command with this bit set, it will not compile this command into the token list under construction, but execute the token immediately. This bit allows structure commands to build special structures in a compound command, and to process special conditions when the compiler is running.

```
 (makeHead) IMMEDIATE call, aanew $80 LIT LAST @ +! ;;
```

```
\ eJ32f.f
: code CODE ;
: ::   CODE ;
```

```
: ;;   return ;

CRR .( Chararter IO ) CRR
:: bye   ext ;;
:: key   get ;;
:: emit  put ;;

CRR .( Common functions ) CRR
:: max ( n n -- n ) 2DUP      < if SWAP then DROP ;;
:: min ( n n -- n ) 2DUP SWAP < if SWAP then DROP ;;
:: /mod ( n n -- r q )
   2DUP / >R MOD R> ;;
:: */ ( n n n -- q )
   >R * R> / ;;

CRR .( Memory access ) CRR
:: execute ( a ) >R ;;
:: here ( -- a ) CP @ ;;
:: pad ( -- a ) CP @ 50 LIT + ;;
:: cmove ( b b u -- )
  for aft >R DUP C@ R@ C! 1+ R> 1+
  then next 2DROP ;;
:: fill ( b u c -- )
  SWAP for SWAP aft 2DUP C! 1+ then next 2DROP ;;

CRR .( Numeric Output ) CRR \ single precision
:: digit ( u -- c )
  9 LIT OVER < 7 LIT AND +
  ( CHAR 0 ) 30 LIT + ;;
:: extract ( n base -- n c )
  /mod SWAP digit ;;
:: <# ( -- ) pad HLD ! ;;
:: hold ( c -- ) HLD @ 1- DUP HLD ! C! ;;
:: # ( u -- u ) BASE @ extract hold ;;
:: #s ( u -- 0 ) begin # DUP while repeat ;;
CRR
:: sign ( n -- ) 0< if ( CHAR - ) 2D LIT hold then ;;
:: #> ( w -- b u ) DROP HLD @ pad OVER - ;;
:: str ( n -- b u ) DUP >R ABS <# #s R> sign #> ;;
:: hex ( -- ) 10 LIT BASE ! ;;
:: decimal ( -- ) 0A LIT BASE ! ;;

CRR .( Basic I/O ) CRR
:: space ( -- ) BL emit ;;
:: spaces ( +n -- ) for aft space then next ;;
:: >char ( c -- c )
  $7E LIT min BL max ;;
:: type ( b u -- )
  for aft COUNT ( >char ) emit
  then next DROP ;;
:: cr ( -- ) ( =Cr )
  0A LIT 0D LIT emit emit ;;
:: do$ ( -- a , get prior frame )
  R> R> DUP COUNT +  >R SWAP >R ;;
```

```
CRR
:: $"| ( -- a ) do$ ;;
:: ."| ( -- ) do$ COUNT type ;;
::  .r ( n +n -- )
  >R str R> OVER - spaces type ;;
::  . ( n -- )
  str space type ;;
:: ? ( a -- ) @ . ;;

CRR .( Numeric Input ) CRR \ single precision
:: digit? ( c base -- u t | x 0 )
  >R ( c ) DUP 40 LIT >
  if 5F LIT AND 37 LIT - ( above @ )
  else DUP 39 LIT >
    if DROP 7F LIT ( above 9 )
    else 30 LIT - DUP ( 0-9 )
      0< if ( below 0 ) DROP 7F LIT then
    then
  then DUP R> ( u u base ) < ( u t | x 0 )
  ;;
:: number? ( a -- n T | a F )
  DUP >R COUNT >R ( a+1 )
  COUNT ( a+2 c - ) 2D LIT = ( a+2 f )
  DUP tmp !
  if R> 1- ( a+2 n-1 )
  else 1- R> ( a+1 n )
  then tmp @ >R iconst_0 tmp !
  for aft ( a' )
    COUNT ( a'+1 c ) BASE @ digit? ( a'+1 b  f )
    if tmp @ BASE @ * + tmp !
    else 2DROP R> R> 2DROP R> iconst_0 ( a 0 ) EXIT
    then
  then next
  DROP tmp @ ( u ) R> if NEGATE then
  iconst_m1 R> DROP ;; ( u t )
CRR .( Parsing ) CRR
:: parse ( c a -- a-1 )
  DUP tmp ! >IN ! ( c )
  begin key DUP emit DUP >IN @ C! $20 LIT > until
  begin iconst_1 >IN +! ( c )
    key DUP emit 2DUP XOR ( c k f1 )
    OVER $1F LIT > ( c k f1 f2 ) AND ( c k f )
  while >IN @ C! ( c )
  repeat ( c k )
  2DROP tmp @ >IN @ OVER - ( a n )
  SWAP 1- SWAP ( a-1 n )
  OVER C! ( a-1 )
  ;;
:: token ( -- a , parser buffer )
  BL CP @ iconst_3 + parse  ;;
:: word ( c -- a , word buffer )
  CP @ 1+ parse ;;
```

```
CRR .( Dictionary Search ) CRR
:: name> ( na -- ca ) COUNT 1F LIT AND +  ;;
:: same? ( a na -- a na diff )
  OVER W@ OVER W@ ( a na ca cna )
  $1FFF ldi AND XOR ucase @ AND ?DUP if EXIT ( a na diff ) then
  OVER C@ 1- >R ( a na )
  OVER 2+ OVER 2+ R> ( a na a+1 na+1 length )
  for aft OVER R@ + C@ ( a na a+i na+i ca )
    OVER R@ + C@ ( a na a+i na+i ca cna )
    XOR ucase @ AND ( a na a+i na+i diff )
    if R> 2DROP ( a na a+i ) EXIT then
  then next ( a na a+i na+i )
  2DROP iconst_0  ;; ( a na 0 )

:: name? ( a -- cfa nfa | a 0 )
  CONTEXT ( a va )
  begin W@ DUP  ( a na na )
  while ( a na )
    same? ( a na f )
    if 2- ( a la ) DUP tmp ! ( save for see )
    else SWAP DROP DUP name> SWAP EXIT ( ca na )
    then
  repeat ;; ( a 0 --, dictionary start )

CRR .( compiler ) CRR
:: [ ( -- )
  iconst_0 COMPI ! ;; IMMEDIATE
:: ] ( -- )
  iconst_m1 COMPI ! ;;
:: , ( n -- ) here DUP CELL+ CP ! ! CP @ DP ! ;;
:: w, ( w -- ) here DUP 2+ CP ! W! CP @ DP ! ;;
:: c, ( c -- ) here DUP 1+ CP ! C! CP @ DP ! ;;
:: allot ( n -- )
   for aft iconst_0 c, then next ;;
:: compil ( w -- ) B6 LIT c, w, ;;
:: literal ( n )
   DUP 0< if CB LIT c, , EXIT then
   DUP $100 ldi < if 10 LIT c, c, EXIT then
   DUP $10000 ldi <
   if 11 LIT c, w,
   else CB LIT c, ,
   then ;;

CRR ( outer interpreter )
:: ok ( -- )
  COMPI @ if else
    cr >R >R >R DUP .
    R> DUP . R> DUP . R> DUP .
    ."| $LIT >ok "
  then ;;
:: quit ( -- )
  [ ( outer interpret )
  begin
    token ( a )
```

```
    name? ( ca na | a 0 )
    ?DUP ( ca na na | a 0 )
    if ( ca na )
      C@ $80 LIT AND ( ca immd )
      if ( ca ) execute
      else
        COMPI @ if compil else execute then
      then
    else ( a )
      number? ( n f | a 0 )
      if ( n ) COMPI @ if literal then
      else ( a )
        DP @ CP ! ( clean dictionary )
        COUNT type $3F LIT emit cr [
      then
    then
    COMPI @ if else ok then
  again
:: abort"| ( f -- )
  if do$ COUNT type quit then do$ DROP ;;
:: error ( a -- )
  space COUNT type $3F LIT emit cr quit

CRR .( colon compiler ) CRR
:: compile ( -- )
  R> 1+ DUP W@ compil
  2+ >R ;;
:: ?unique ( a -- a )
  DUP name?
  if COUNT type ."| $LIT reDef "
  then DROP ;;
:: $,n ( a -- )
  DUP @
  if ?unique
    ( na) CP @ DP !
    ( na) DUP name> CP !
    ( na) DUP LAST ! \ for overt
    ( na) 2-
    ( la) CONTEXT W@ SWAP W! EXIT
  then error
:: overt ( -- ) LAST @ CONTEXT W! ;;
:: ; ( -- )
  B1 LIT c, [ overt ;; IMMEDIATE
:: : ( -- ; <string> )
  token $,n ] ;;

CRR .( Tools ) CRR
:: ' ( -- ca )
  token name? if EXIT then
  error
:: dm+ ( b u -- b+u )
  OVER 6 LIT .r space
  for aft DUP C@ 3 LIT .r 1+
  then next ;;
```

```
:: dump ( b u -- )
  hex  10 LIT /
  for aft cr 10 LIT dm+ space
    DUP 10 LIT - 10 LIT type
  then next DROP ;;

CRR
:: >name ( ca -- na | F )
  CONTEXT ( ca la )
  begin W@ DUP ( ca na na )
  while 2DUP name> ( ca na ca ca ) XOR
    if 2- ( ca la )
    else SWAP DROP EXIT ( na )
    then
  repeat SWAP DROP ( na=0 ) ;;
:: id ( a -- )
  COUNT $01F LIT AND type space ;;

CRR
:: see ( -- ; <string> )
  cr ' ( ca --, tmp has next la )
  begin ( ca )
    COUNT DUP $B6 LIT XOR ( ca+1 b f )
    if . space
    else DROP COUNT >R ( ca+2 )
      COUNT $100 ldi * R> + ( ca+3 w ) >name
      ?DUP if id else $5F LIT emit space then
      1+ ( ca+4 )
    then
    DUP tmp @ > ( ca+4 )
  until DROP ;;
:: words ( -- )
  cr CONTEXT
  begin W@ ?DUP
  while DUP id 2-
  repeat cr ;;
:: case if $FFFFFFDF ldi else $FFFFFFFF ldi then ucase ! ;;

CRR .( Hardware reset ) CRR
:: diagnose     ( - )
     get put get put get put
     9 LIT 9 LIT =
     8 LIT 9 LIT =
     9 LIT 8 LIT =
     $65 LIT iconst_0 0< +
\     mask
\ 'F'  prove + 0<
     -2 ldi 0<     \ -1
     4 LIT +     \ 3
     $43 LIT +       \ 'F'
\ 'o' logic: XOR AND OR
     $4F LIT $6F LIT XOR    \ 20h
     $F0 LIT AND
     $4F LIT OR
```

```
\ 'r' stack: DUP OVER SWAP DROP
      8 LIT 6 LIT SWAP
      OVER XOR 3 LIT AND AND
      $70 LIT +        \ 'r'
\ 't'-- prove BRANCH ?BRANCH
      iconst_0 if $3F LIT then
      -1 ldi if $74 LIT else $21 LIT then
\ 'h' -- @ ! test memeory address
      $68 LIT $30 LIT !
      $30 LIT @
\ 'M' -- prove >R R> R@
      $4D LIT >R R@ R> AND
\ 'l'  -- prove 'next' can run
      $61 LIT $A LIT for iconst_1 + next
\ 'S' -- prove 2!, 2@
      $50 LIT $3 LIT
      $30 LIT 2!
      $30 LIT 2@
      XOR
\ 'em' -- prove imul, idiv, irem
      $656D ldi $10000 ldi *
      DUP $1000000 ldi /
      swap $1000000 ldi MOD
      $10 LIT ishr
\ 'it' -- prove /, MOD
      $7469 ldi DUP $100 ldi MOD
      swap $100 ldi /
      ;;

:: cold ( -- )
   diagnose
   cr ."| $LIT eJ32 v1.01"
   cr quit

CRR ( conditionals ) CRR
(makehead) 0<
   ifneg iconst_0 else iconst_m1 then
   return
(makehead) =
   ifeqq iconst_0 else iconst_m1 then
   return
(makehead) >
   ifgreat iconst_0 else iconst_m1 then
   return
(makehead) <
   ifless iconst_0 else iconst_m1 then
   return
(makehead) ?dup
   dup if dup then return
(makehead) abs
   dup ifneg else ineg then return

CRR .( Structures ) CRR
:: begin ( -- a ) here ;; IMMEDIATE
```

```
:: then  ( A -- ) begin SWAP W! ;; IMMEDIATE
:: for   ( -- a ) $CD LIT c, begin ;; IMMEDIATE
CRR
:: next  ( a -- ) $CA LIT c, w, ;; IMMEDIATE
:: until ( a -- ) $99 LIT c, w, ;; IMMEDIATE
:: again ( a -- ) $A7 LIT c, w, ;; IMMEDIATE
:: if    ( -- A ) $99 LIT c, begin 0 LIT w, ;; IMMEDIATE
:: ahead ( -- A ) $A7 LIT c, begin 0 LIT w, ;; IMMEDIATE
CRR
:: repeat ( A a -- ) again then ;; IMMEDIATE
:: aft  ( a -- a A ) DROP ahead begin SWAP ;; IMMEDIATE
:: else   ( A -- A ) ahead SWAP then ;; IMMEDIATE
:: while ( a -- A a) if SWAP ;; IMMEDIATE

CRR ( strings ) CRR
:: $," ( -- ) ( CHAR " )
   $22 LIT word DUP C@ + 1+ CP ! ;;
:: abort" ( -- ; <string> )
   $B6 LIT c, forth_' abort"| >body forth_@ ldi w,
   $," ;; IMMEDIATE
:: $" ( -- ; <string> )
   $B6 LIT c, forth_' $"| >body forth_@ ldi w,
   $," ;; IMMEDIATE
:: ." ( -- ; <string> )
   $B6 LIT c, forth_' ."| >body forth_@ ldi w,
   $," ;; IMMEDIATE

CRR ( defining ) CRR
:: docon popr @ ;;
:: dovar popr ;;
:: does ( -- ) popr LAST @ name> 1+ ! ;;
:: code ( -- ; <string> )
   token $,n overt ;;
:: create ( -- ; <string> )
   code $B6 LIT c, forth_' dovar >body forth_@ ldi w,
   CP @ DP ! ;;
:: variable ( -- ; <string> )
   create 0 LIT , CP @ DP ! ;;
:: constant ( n --; <string> )
   code $B6 LIT c, forth_' docon >body forth_@ ldi w,
   , CP @ DP ! ;;

CRR
(makehead) r> $CC LIT c, return IMMEDIATE
(makehead) r@ $CE LIT c, return IMMEDIATE
(makehead) >r $CD LIT c, return IMMEDIATE
(makehead) .( ( -- ) $29 LIT parse type ;; IMMEDIATE
(makehead) ( $29 LIT parse 2DROP ;; IMMEDIATE
(makehead) immediate $80 LIT LAST +! ;;

CRR ( primitives ) CRR
(makehead) exit return
(makehead) ! swap iastore return
(makehead) @ iaload return
```

```
(makehead) c! swap bastore return
(makehead) c@ baload return
(makehead) w! swap sastore return
(makehead) w@ saload return

(makehead) swap swap return
(makehead) drop pop return
(makehead) 2drop pop2 return
(makehead) over dup2 pop return
(makehead) 2dup dup2 return

(makehead) +  iadd return
(makehead) -  isub return
(makehead) *  imul return
(makehead) /  idiv return
(makehead) mod  irem return
(makehead) or  ior return
(makehead) and  iand return
(makehead) xor  ixor return
(makehead) not iconst_m1 ixor return
(makehead) negate ineg return
(makehead) 1- iconst_m1 iadd return
(makehead) 1+ iconst_1 iadd return
(makehead) 2- iconst_2 isub return
(makehead) 2+ iconst_2 iadd return
(makehead) cell- iconst_4 isub return
(makehead) cell+ iconst_4 iadd return

(makehead) bl $20 LIT return
(makehead) +!
   dup pushr iaload iadd
   popr swap iastore return
(makehead) rot
   pushr swap popr swap return
(makehead) -rot
   dup_x2 pop return
(makehead) 2!
   dup2 swap iastore swap pop
   iconst_4 iadd swap iastore return
(makehead) 2@
   dup iaload swap iconst_4 iadd iaload swap
   return
(makehead) count
   dup baload swap iconst_1 iadd swap return
(makehead) dup dup return

CRR

h forth_@ forth_dup forth_dup

0 org
cold 0 #, 0 #, 0 #, 0 #,

$40 org
```

```
$10 #,
$0  #,
$0  #,
$0  #,
lasth forth_@ $10 LSHIFT #,
( h )         #,
( h )         #,
( h )         #,
$0  #,
$0  #,
$FFFFFFDF  #,
$1000 #,
$1400 #,
```

eJ32Isim.f

```
 D0B1 forth_' key  >body forth_@ ramw!
 D1B1 forth_' emit >body forth_@ ramw!

forth_forget h

DECIMAL
$3F CONSTANT LIMIT ( stack depth )
$1FFF CONSTANT RANGE ( program memory size in byte )
VARIABLE CLOCK
VARIABLE (REGISTER) ( where registers and stacks are )
VARIABLE BREAK
VARIABLE input $1000 input !
VARIABLE output $1400 output !

: REGISTER  PAD ;

: P     REGISTER ;
: RP    REGISTER 8 + ;
: SP    REGISTER 12 + ;
: T     REGISTER 16 + ;
: RSTACK  RP C@ LIMIT AND 4 * REGISTER + $100 + ;
: SSTACK  SP C@ LIMIT AND 4 * REGISTER + $200 + ;
: S  SSTACK ;
: R  RSTACK ;

: RPUSH ( n -- , push n on return stack )
       4 RP +! RSTACK ! ;
: RPOPP ( -- n , pop n from return stack )
       RSTACK @ -4 RP +! ;
: SPUSH ( n -- , push n on data stack )
       T @ 1 SP +! SSTACK ! T ! ;
: SPOPP ( -- n , pop n from data stack )
       T @ SSTACK @ T ! -1 SP +! ;
: CYCLE 1 CLOCK +! ;

: continue 1 P +! ;
: JUMP  2 P +! ;
: bra   P @ 1+ ramw@ 1- P ! ;
: bz    SPOPP IF JUMP ELSE bra THEN ;
: call  P @ 3 + RPUSH bra ;
: return R @ 1- P ! -4 RP +!  ;
: ret   R P @ 1+ ramc@ CELLS - @ 1- P ! ;

: get   KEY DUP $1B = ABORT" done"
       SPUSH ;
```

```
: put   SPOPP $7F AND EMIT ;

DECIMAL

: execute ( code -- )
 DUP 00 = ( 0x00 nop ) IF DROP EXIT THEN
 DUP 01 = ( 0x01 aconst_null ) IF DROP 0 SPUSH EXIT THEN
 DUP 02 = ( 0x02 iconst_m1 ) IF DROP -1 SPUSH EXIT THEN
 DUP 03 = ( 0x03 iconst_0 ) IF DROP 0 SPUSH EXIT THEN
 DUP 04 = ( 0x04 iconst_1 ) IF DROP 1 SPUSH EXIT THEN
 DUP 05 = ( 0x05 iconst_2 ) IF DROP 2 SPUSH EXIT THEN
 DUP 06 = ( 0x06 iconst_3 ) IF DROP 3 SPUSH EXIT THEN
 DUP 07 = ( 0x07 iconst_4 ) IF DROP 4 SPUSH EXIT THEN
 DUP 08 = ( 0x08 iconst_5 ) IF DROP 5 SPUSH EXIT THEN
 DUP 09 = ( 0x09 lconst_0 ) IF DROP EXIT THEN
 DUP 10 = ( 0x0a lconst_1 ) IF DROP EXIT THEN
 DUP 11 = ( 0x0b fconst_0 ) IF DROP EXIT THEN
 DUP 12 = ( 0x0c fconst_1 ) IF DROP EXIT THEN
 DUP 13 = ( 0x0d fconst_2 ) IF DROP EXIT THEN
 DUP 14 = ( 0x0e dconst_0 ) IF DROP EXIT THEN
 DUP 15 = ( 0x0f dconst_1 ) IF DROP EXIT THEN
 DUP 16 = ( 0x10 bipush ) IF DROP P @ 1+ RAMC@ DUP $80 AND IF
$FFFFFF00 + THEN
     SPUSH 1 P +! EXIT THEN
 DUP 17 = ( 0x11 sipush ) IF DROP p @ 1+ RAMW@ DUP $8000 AND IF
$FFFF0000 + THEN
     SPUSH 2 P +! EXIT THEN
 DUP 18 = ( 0x12 ldc ) IF DROP EXIT THEN
 DUP 19 = ( 0x13 ldc_w ) IF DROP EXIT THEN
 DUP 20 = ( 0x14 ldc2_w ) IF DROP EXIT THEN
 DUP 21 = ( 0x15 iload ) IF DROP R P @ 1+ ramc@ 4 *
     - @ SPUSH 1 P +! EXIT THEN
 DUP 22 = ( 0x16 lload ) IF DROP EXIT THEN
 DUP 23 = ( 0x17 fload ) IF DROP EXIT THEN
 DUP 24 = ( 0x18 dload ) IF DROP EXIT THEN
 DUP 25 = ( 0x19 aload ) IF DROP EXIT THEN
 DUP 26 = ( 0x1a iload_0 ) IF DROP R @ SPUSH EXIT THEN
 DUP 27 = ( 0x1b iload_1 ) IF DROP R 4 - @ SPUSH EXIT THEN
 DUP 28 = ( 0x1c iload_2 ) IF DROP R 8 - @ SPUSH EXIT THEN
 DUP 29 = ( 0x1d iload_3 ) IF DROP R 12 - @ SPUSH EXIT THEN
 DUP 30 = ( 0x1e lload_0 ) IF DROP EXIT THEN
 DUP 31 = ( 0x1f lload_1 ) IF DROP EXIT THEN
 DUP 32 = ( 0x20 lload_2 ) IF DROP EXIT THEN
 DUP 33 = ( 0x21 lload_3 ) IF DROP EXIT THEN
 DUP 34 = ( 0x22 fload_0 ) IF DROP EXIT THEN
 DUP 35 = ( 0x23 fload_1 ) IF DROP EXIT THEN
 DUP 36 = ( 0x24 fload_2 ) IF DROP EXIT THEN
```

```
DUP 37 = ( 0x25 fload_3 ) IF DROP EXIT THEN
DUP 38 = ( 0x26 dload_0 ) IF DROP EXIT THEN
DUP 39 = ( 0x27 dload_1 ) IF DROP EXIT THEN
DUP 40 = ( 0x28 dload_2 ) IF DROP EXIT THEN
DUP 41 = ( 0x29 dload_3 ) IF DROP EXIT THEN
DUP 42 = ( 0x2a aload_0 ) IF DROP EXIT THEN
DUP 43 = ( 0x2b aload_1 ) IF DROP EXIT THEN
DUP 44 = ( 0x2c aload_2 ) IF DROP EXIT THEN
DUP 45 = ( 0x2d aload_3 ) IF DROP EXIT THEN
DUP 46 = ( 0x2e iaload ) IF DROP SPOPP ram@ SPUSH EXIT THEN
DUP 47 = ( 0x2f laload ) IF DROP EXIT THEN
DUP 48 = ( 0x30 faload ) IF DROP EXIT THEN
DUP 49 = ( 0x31 daload ) IF DROP EXIT THEN
DUP 50 = ( 0x32 aaload ) IF DROP EXIT THEN
DUP 51 = ( 0x33 baload ) IF DROP SPOPP ramc@ SPUSH EXIT THEN
DUP 52 = ( 0x34 caload ) IF DROP EXIT THEN
DUP 53 = ( 0x35 saload ) IF DROP SPOPP ramw@ SPUSH EXIT THEN
DUP 54 = ( 0x36 istore ) IF DROP SPOPP R P @ 1+ ramc@ 4 *
    - ! 1 P +! EXIT THEN
DUP 55 = ( 0x37 lstore ) IF DROP EXIT THEN
DUP 56 = ( 0x38 fstore ) IF DROP EXIT THEN
DUP 57 = ( 0x39 dstore ) IF DROP EXIT THEN
DUP 58 = ( 0x3a astore ) IF DROP EXIT THEN
DUP 59 = ( 0x3b istore_0 ) IF DROP SPOPP R ! EXIT THEN
DUP 60 = ( 0x3c istore_1 ) IF DROP SPOPP R 4 - ! EXIT THEN
DUP 61 = ( 0x3d istore_2 ) IF DROP SPOPP R 8 - ! EXIT THEN
DUP 62 = ( 0x3e istore_3 ) IF DROP SPOPP R 12 - ! EXIT THEN
DUP 63 = ( 0x3f lstore_0 ) IF DROP EXIT THEN
DUP 64 = ( 0x40 lstore_1 ) IF DROP EXIT THEN
DUP 65 = ( 0x41 lstore_2 ) IF DROP EXIT THEN
DUP 66 = ( 0x42 lstore_3 ) IF DROP EXIT THEN
DUP 67 = ( 0x43 fstore_0 ) IF DROP EXIT THEN
DUP 68 = ( 0x44 fstore_1 ) IF DROP EXIT THEN
DUP 69 = ( 0x45 fstore_2 ) IF DROP EXIT THEN
DUP 70 = ( 0x46 fstore_3 ) IF DROP EXIT THEN
DUP 71 = ( 0x47 dstore_0 ) IF DROP EXIT THEN
DUP 72 = ( 0x48 dstore_1 ) IF DROP EXIT THEN
DUP 73 = ( 0x49 dstore_2 ) IF DROP EXIT THEN
DUP 74 = ( 0x4a dstore_3 ) IF DROP EXIT THEN
DUP 75 = ( 0x4b astore_0 ) IF DROP EXIT THEN
DUP 76 = ( 0x4c astore_1 ) IF DROP EXIT THEN
DUP 77 = ( 0x4d astore_2 ) IF DROP EXIT THEN
DUP 78 = ( 0x4e astore_3 ) IF DROP EXIT THEN
DUP 79 = ( 0x4f iastore ) IF DROP SPOPP SPOPP ram! EXIT THEN
DUP 80 = ( 0x50 lastore ) IF DROP EXIT THEN
DUP 81 = ( 0x51 fastore ) IF DROP EXIT THEN
DUP 82 = ( 0x52 dastore ) IF DROP EXIT THEN
```

```
 DUP 83 = ( 0x53 aastore ) IF DROP EXIT THEN
 DUP 84 = ( 0x54 bastore ) IF DROP SPOPP SPOPP RAMC! EXIT THEN
 DUP 85 = ( 0x55 castore ) IF DROP EXIT THEN
 DUP 86 = ( 0x56 sastore ) IF DROP SPOPP SPOPP RAMW! EXIT THEN
 DUP 87 = ( 0x57 pop ) IF DROP SPOPP DROP EXIT THEN
 DUP 88 = ( 0x58 pop2 ) IF DROP SPOPP DROP SPOPP DROP EXIT THEN
 DUP 89 = ( 0x59 dup ) IF DROP T @ SPUSH EXIT THEN
 DUP 90 = ( 0x5a dup_x1 ) IF DROP SPOPP SPOPP OVER SPUSH SPUSH
SPUSH EXIT THEN
 DUP 91 = ( 0x5b dup_x2 ) IF DROP SPOPP SPOPP OVER SPOPP SWAP
     SPUSH SPUSH SPUSH SPUSH EXIT THEN
 DUP 92 = ( 0x5c dup2 ) IF DROP SPOPP SPOPP OVER OVER SPUSH
SPUSH SPUSH SPUSH EXIT THEN
 DUP 93 = ( 0x5d dup2_x1 ) IF DROP EXIT THEN
 DUP 94 = ( 0x5e dup2_x2 ) IF DROP EXIT THEN
 DUP 95 = ( 0x5f swap ) IF DROP SPOPP SPOPP SWAP SPUSH SPUSH
EXIT THEN
 DUP 96 = ( 0x60 iadd ) IF DROP SPOPP SPOPP + SPUSH EXIT THEN
 DUP 97 = ( 0x61 ladd ) IF DROP EXIT THEN
 DUP 98 = ( 0x62 fadd ) IF DROP EXIT THEN
 DUP 99 = ( 0x63 dadd ) IF DROP EXIT THEN
 DUP 100 = ( 0x64 isub ) IF DROP SPOPP SPOPP SWAP - SPUSH EXIT
THEN
 DUP 101 = ( 0x65 lsub ) IF DROP EXIT THEN
 DUP 102 = ( 0x66 fsub ) IF DROP EXIT THEN
 DUP 103 = ( 0x67 dsub ) IF DROP EXIT THEN
 DUP 104 = ( 0x68 imul ) IF DROP SPOPP SPOPP * SPUSH EXIT THEN
 DUP 105 = ( 0x69 lmul ) IF DROP EXIT THEN
 DUP 106 = ( 0x6a fmul ) IF DROP EXIT THEN
 DUP 107 = ( 0x6b dmul ) IF DROP EXIT THEN
 DUP 108 = ( 0x6c idiv ) IF DROP SPOPP SPOPP SWAP / SPUSH EXIT
THEN
 DUP 109 = ( 0x6d ldiv ) IF DROP EXIT THEN
 DUP 110 = ( 0x6e fdiv ) IF DROP EXIT THEN
 DUP 111 = ( 0x6f ddiv ) IF DROP EXIT THEN
 DUP 112 = ( 0x70 irem ) IF DROP SPOPP SPOPP SWAP MOD SPUSH EXIT
THEN
 DUP 113 = ( 0x71 lrem ) IF DROP EXIT THEN
 DUP 114 = ( 0x72 frem ) IF DROP EXIT THEN
 DUP 115 = ( 0x73 drem ) IF DROP EXIT THEN
 DUP 116 = ( 0x74 ineg ) IF DROP SPOPP NEGATE SPUSH EXIT THEN
 DUP 117 = ( 0x75 lneg ) IF DROP EXIT THEN
 DUP 118 = ( 0x76 fneg ) IF DROP EXIT THEN
 DUP 119 = ( 0x77 dneg ) IF DROP EXIT THEN
 DUP 120 = ( 0x78 ishl ) IF DROP SPOPP SPOPP SWAP LSHIFT SPUSH
EXIT THEN
 DUP 121 = ( 0x79 lshl ) IF DROP EXIT THEN
```

```
 DUP 122 = ( 0x7a ishr ) IF DROP SPOPP SPOPP SWAP RSHIFT SPUSH
EXIT THEN
 DUP 123 = ( 0x7b lshr ) IF DROP EXIT THEN
 DUP 124 = ( 0x7c iushr ) IF DROP EXIT THEN
 DUP 125 = ( 0x7d lushr ) IF DROP EXIT THEN
 DUP 126 = ( 0x7e iand ) IF DROP SPOPP SPOPP AND SPUSH EXIT THEN
 DUP 127 = ( 0x7f land ) IF DROP EXIT THEN
 DUP 128 = ( 0x80 ior ) IF DROP SPOPP SPOPP OR SPUSH EXIT THEN
 DUP 129 = ( 0x81 lor ) IF DROP EXIT THEN
 DUP 130 = ( 0x82 ixor ) IF DROP SPOPP SPOPP XOR SPUSH EXIT THEN
 DUP 131 = ( 0x83 lxor ) IF DROP EXIT THEN
 DUP 132 = ( 0x84 iinc ) IF DROP SPOPP SPOPP DUP ram@ ROT + SWAP
ram! EXIT THEN
 DUP 133 = ( 0x85 i2l ) IF DROP EXIT THEN
 DUP 134 = ( 0x86 i2f ) IF DROP EXIT THEN
 DUP 135 = ( 0x87 i2d ) IF DROP EXIT THEN
 DUP 136 = ( 0x88 l2i ) IF DROP EXIT THEN
 DUP 137 = ( 0x89 l2f ) IF DROP EXIT THEN
 DUP 138 = ( 0x8a l2d ) IF DROP EXIT THEN
 DUP 139 = ( 0x8b f2i ) IF DROP EXIT THEN
 DUP 140 = ( 0x8c f2l ) IF DROP EXIT THEN
 DUP 141 = ( 0x8d f2d ) IF DROP EXIT THEN
 DUP 142 = ( 0x8e d2i ) IF DROP EXIT THEN
 DUP 143 = ( 0x8f d2l ) IF DROP EXIT THEN
 DUP 144 = ( 0x90 d2f ) IF DROP EXIT THEN
 DUP 145 = ( 0x91 i2b ) IF DROP EXIT THEN
 DUP 146 = ( 0x92 i2c ) IF DROP EXIT THEN
 DUP 147 = ( 0x93 i2s ) IF DROP EXIT THEN
 DUP 148 = ( 0x94 lcmp ) IF DROP EXIT THEN
 DUP 149 = ( 0x95 fcmpl ) IF DROP EXIT THEN
 DUP 150 = ( 0x96 fcmpg ) IF DROP EXIT THEN
 DUP 151 = ( 0x97 dcmpl ) IF DROP EXIT THEN
 DUP 152 = ( 0x98 dcmpg ) IF DROP EXIT THEN
 DUP 153 = ( 0x99 ifeq ) IF DROP SPOPP IF JUMP ELSE bra THEN
EXIT THEN
 DUP 154 = ( 0x9a ifne ) IF DROP SPOPP IF bra ELSE JUMP THEN
EXIT THEN
 DUP 155 = ( 0x9b iflt ) IF DROP SPOPP 0< IF bra ELSE JUMP THEN
EXIT THEN
 DUP 156 = ( 0x9c ifge ) IF DROP SPOPP 0< IF JUMP ELSE bra THEN
EXIT THEN
 DUP 157 = ( 0x9d ifgt ) IF DROP SPOPP 0 > IF bra ELSE JUMP THEN
EXIT THEN
 DUP 158 = ( 0x9e ifle ) IF DROP SPOPP 0 > IF JUMP ELSE bra THEN
EXIT THEN
 DUP 159 = ( 0x9f if_icmpeq ) IF DROP SPOPP SPOPP = IF bra ELSE
JUMP THEN EXIT THEN
```

```
 DUP 160 = ( 0xa0 if_icmpne ) IF DROP EXIT THEN
 DUP 161 = ( 0xa1 if_icmplt ) IF DROP SPOPP SPOPP SWAP < IF bra
ELSE JUMP THEN EXIT THEN
 DUP 162 = ( 0xa2 if_icmpge ) IF DROP EXIT THEN
 DUP 163 = ( 0xa3 if_icmpgt ) IF DROP SPOPP SPOPP SWAP > IF bra
ELSE JUMP THEN EXIT THEN
 DUP 164 = ( 0xa4 if_icmple ) IF DROP EXIT THEN
 DUP 165 = ( 0xa5 if_acmpeq ) IF DROP EXIT THEN
 DUP 166 = ( 0xa6 if_acmpne ) IF DROP EXIT THEN
 DUP 167 = ( 0xa7 goto ) IF DROP P @ 1+ RAMW@ 1- P ! EXIT THEN
 DUP 168 = ( 0xa8 jsr ) IF DROP bra P @ 2+ SPUSH EXIT THEN
 DUP 169 = ( 0xa9 ret ) IF DROP ret EXIT THEN
 DUP 170 = ( 0xaa tableswitch ) IF DROP EXIT THEN
 DUP 171 = ( 0xab lookupswitch ) IF DROP EXIT THEN
 DUP 172 = ( 0xac ireturn ) IF DROP EXIT THEN
 DUP 173 = ( 0xad lreturn ) IF DROP EXIT THEN
 DUP 174 = ( 0xae freturn ) IF DROP EXIT THEN
 DUP 175 = ( 0xaf dreturn ) IF DROP EXIT THEN
 DUP 176 = ( 0xb0 areturn ) IF DROP EXIT THEN
 DUP 177 = ( 0xb1 return ) IF DROP return EXIT THEN
 DUP 178 = ( 0xb2 getstatic ) IF DROP EXIT THEN
 DUP 179 = ( 0xb3 putstatic ) IF DROP EXIT THEN
 DUP 180 = ( 0xb4 getfield ) IF DROP EXIT THEN
 DUP 181 = ( 0xb5 putfield ) IF DROP EXIT THEN
 DUP 182 = ( 0xb6 invokevirtual ) IF DROP call EXIT THEN
 DUP 183 = ( 0xb7 invokespecial ) IF DROP EXIT THEN
 DUP 184 = ( 0xb8 invokestatic ) IF DROP EXIT THEN
 DUP 185 = ( 0xb9 invokeinterface ) IF DROP EXIT THEN
 DUP 186 = ( 0xba invokedynamic ) IF DROP EXIT THEN
 DUP 187 = ( 0xbb new ) IF DROP EXIT THEN
 DUP 188 = ( 0xbc newarray ) IF DROP EXIT THEN
 DUP 189 = ( 0xbd anewarray ) IF DROP EXIT THEN
 DUP 190 = ( 0xbe arraylength ) IF DROP EXIT THEN
 DUP 191 = ( 0xbf athrow ) IF DROP EXIT THEN
 DUP 192 = ( 0xc0 checkcast ) IF DROP EXIT THEN
 DUP 193 = ( 0xc1 instanceof ) IF DROP EXIT THEN
 DUP 194 = ( 0xc2 monitorenter ) IF DROP EXIT THEN
 DUP 195 = ( 0xc3 monitorexit ) IF DROP EXIT THEN
 DUP 196 = ( 0xc4 wide ) IF DROP EXIT THEN
 DUP 197 = ( 0xc5 multianewarray ) IF DROP EXIT THEN
 DUP 198 = ( 0xc6 ifnull ) IF DROP EXIT THEN
 DUP 199 = ( 0xc7 ifnonnull ) IF DROP EXIT THEN
 DUP 200 = ( 0xc8 goto_w ) IF DROP EXIT THEN
 DUP 201 = ( 0xc9 jsr_w ) IF DROP EXIT THEN
 DUP 202 = ( 0xca donext ) IF DROP
     R -1 OVER +! @ 0< IF -4 RP +! JUMP ELSE bra THEN EXIT THEN
```

```
 DUP 203 = ( 0xcb ldi ) IF DROP P @ 1+ RAM@ SPUSH 4 P +! EXIT
THEN
 DUP 204 = ( 0xcc popr ) IF DROP RPOPP SPUSH EXIT THEN
 DUP 205 = ( 0xcd pushr ) IF DROP SPOPP RPUSH EXIT THEN
 DUP 206 = ( 0xce dupr ) IF DROP R @ SPUSH EXIT THEN
\ DUP 207 = ( 0xcf ext ) IF DROP EXIT THEN
\ DUP 208 = ( 0xd0 get ) IF DROP KEY SPUSH EXIT THEN
\ DUP 209 = ( 0xd1 put ) IF DROP SPOPP EMIT EXIT THEN
 DUP 208 = ( 0xd0 get ) IF DROP  input @ ramc@ SPUSH 1  input +!
EXIT THEN
 DUP 209 = ( 0xd1 put ) IF DROP SPOPP output @ ramc! 1 output +!
EXIT THEN
 . -1 ABORT" : Illegel instruction" ;
HEX
: .stack ( add # ) FOR AFT DUP @ U. 4 - THEN NEXT DROP CR ;
: .sstack ." S:" T ? SSTACK SP C@ .stack ;
: .rstack ." R:" RSTACK RP C@ .stack ;
: .registers ."  P=" P @ DUP . ."  code=" ram@ . CR ;
: S  CR ." CLOCK=" CLOCK @ . .registers .sstack .rstack ;
: exec  P @ ramc@ execute 1 P +! ;
: C     exec CYCLE S .S ;
: reset P $300 0 FILL 0 CLOCK ! ;
reset

: G     ( addr -- )
        CR ." Press any key to stop." CR
        BREAK !
        BEGIN exec P @ BREAK @ =
              IF CYCLE C EXIT
              ELSE CYCLE
              THEN
              ?KEY
        UNTIL ;

: D     P @ 1- FOUR ;
: M     show ;
: RUN   CR ." Press ESC to stop." CR
        BEGIN C KEY 1B = UNTIL ;
\ : P     DUP RANGE AND P ! RANGE AND P ! ;

: HELP  CR ." eP32 Simulator, copyright eForth Group, 2000"
        CR ." C: execute next cycle"
        CR ." S: show all registers"
        CR ." D: display next 8 words"
        CR ." addr M: display 128 words from addr"
        CR ." addr P: start execution at addr"
        CR ." addr G: run and stop at addr"
```

```
        CR ." RUN: execute, one key per cycle"
        CR ;
HELP
 -1 G
```

Chapter 7. Implementation Notes

Byte Code Sequencer vs Finite State Machine

A Finite State Machine (FSM) was adopted from eP32 chip design to run VFM in ceForth. This FSM assumed that we had a 32 bit machine, running on 32 bit memory. It used 6 phases to execute code stored in memory. In phase 0, it read a 32 bit program word, and decoded 4 byte code in it. In phase 1 to 4 it executes these 4 byte code in sequence. In phase 5, it resets the phase counter to 0, so it will fetch the next program word from memory, and run through the phases again. This FSM is described completely in the `loop()` routine:

```
void loop() {
    phase = clk & 7;
    switch(phase) {
      case 0: fetch_decode(); yield(); break;
      case 1: execute(I1); break;
      case 2: execute(I2); break;
      case 3: execute(I3); break;
      case 4: execute(I4); break;
      case 5: jump(); break;
      case 6: jump(); break;
      case 7: jump();
    }
    clk += 1;
}
```

In JFM, the dictionary is an array of 32 bit words. However, this array can be read either in 32 bit words, or in 8 bit bytes. Therefore, byte code in the dictionary can be fetched directly and executed without a FSM. A much simpler byte code sequencer can be coded as follows:

```
void loop() {
  while (TRUE) {
    bytecode = (unsigned char)cData[P++];
    execute(bytecode);
  } }
```

The sequence has only two steps: fetching next byte from memory, and execute the byte code. It is just like a hardware computer, sequencing through its memory to execute machine instructions.

In the design of JFM VFM, byte code are packed into code fields of primitive commands, and can be accessed either by 32 bit words, or by byte sequence. The same dictionary accommodates both design equally well. No modification in ceForth dictionary is necessary.

Stacks

Stacks are big headaches in operating systems, and in application programs. In C programming, stacks are hidden from you to prevent you from messing them up. However, in Forth programming, the data stack and the return stacks are open to you, and most of the times, the data stack becomes the focus of your attention. Both stacks have to work perfectly. There is no margin of error.

With stacks implemented in memory of finite size, the most obvious problems are stack overflow and stack underflow. Generally, operating systems allocate large chucks of memory for stacks, and impose traps on overflow and underflow conditions. With these traps, you can write interrupt routines to handle these error conditions in your software. These traps are very difficult to handle, especially for those without advanced computer science degrees.

The most prevalent problem in Forth programming is underflow of data stack, when you try to access data below the memory allocated to data stack. After Forth interpreter finished interpreting a sequence of Forth words, it always check the stack pointer. If the stack point is below mark, Forth interpreter executes the ABORT command, and reinitialized the stacks.

In designing eP32 chip, I put both stacks in the CPU. I allowed 32 levels of stack space, and the system seems to be happy. I checked often the water marks on both stacks, and the water marks were mostly about 12 levels. 32 levels are adequate for most applications, and do not impose a big burden on CPU designs. The stacks used 5 bit stack pointers, and behaved like circular buffers. I also found that it was not really necessary to check the stack pointers. Using circular buffers, underflow and overflow are really not life-threatening error conditions. If useful data were actually overwritten, the system would not behave correctly, but in no danger of crashing. The stack pointers need not be reset. The system would restart with the present pointers.

In JFM_44, I allocated 1KB memory for each stack, and used one byte for each stack pointer. The stacks are 256 cell circular buffers, and will never underflow or overflow. However, the C compiler needs to be reminder constantly that the stack pointers have 8-bit values and must not be leaked to integer or long number. R and S pointers must always be prefixed with `(unsigned char)` specification. I struggled with data stack underflow conditions for half a year, until I found that the stack pointers tended to overshot the byte boundary in my back.

JFM interpreter always displays top 4 elements on data stack. Always seeing these 4 elements, you do not need utility to dump or examine data stack. I believe this is the best way to use data stack, and relieve you from the anxiety of worrying your misusing it.

MetaCompiler

Conceptually, metacompilation is not much different that the ordinary Forth compiler. Forth compiler compiles new commands on top of its dictionary. CP is the pointer to top of dictionary. If we change CP to point to another memory location, like the target dictionary array we allocated for a target system, then we can compile a new dictionary for the target.

Of course, the devil is in the details. The target memory is a virtual memory. Addresses used by the target machine are virtual addresses relative to the beginning of the dictionary array, not the absolute addresses used in the host Forth system. The target machine may have a different machine instruction set. Byte addressable machine vs word addressable machine. Different linking schemes. On and on.

The art of metacompilation had been practices since Chuck Moore invented Forth. I documented it for polyForth, F83, and FPC, three of the most popular Forth implementations. They all used vocabularies to segregate names of same commands used at various stages of metacompilation. For example, + (plus) command had 3 different behaviors: a regular + (plus) version to add two integers in text interpreter, a version defined in target dictionary which will be used by a target system to add two integers, which is never executed during metacompilation, and one version used by metacompiler to compile a + (plus) token in the body (token list) of a compound command in tart dictionary.

A dictionary is a linked list of command records. A vocabulary is a branch of a dictionary, which can be searched independent of the main dictionary. Vocabularies allow a command to be redefined multiple times, and different behavior is selected by specifying search order of vocabularies.

In the original eForth Model, Bill Muench reserved system variables to allow building up to 8 vocabularies. However, over the years I had not used this feature in all my applications, and decided to rid of it. Without vocabularies, I could still do metacompilation by carefully arranging the sequence in defining commands to build target dictionary correctly. As commands are redefined, the Forth system morphs and shows unexpected behavior. All eForth commands are redefined to compile tokens. At the end of metacompilation, you can type in any valid eForth command, and the system responds with ‘ok’, but does not seem to do anything. The data stack does not change. All the commands do is to add a token to the top of target dictionary, which you cannot observed without great efforts.

After the metacompiler finishes building the target dictionary, it is useless for any other purposes. Close F# and use the eJ32i.mif file to build JFM on Arduino IDE.

Byte Code

I use byte code to bridge the JVM and the eForth system. The dictionary is stored is an integer array `data[].` This data array can be addressed either by 32 bit words or by bytes. When addressing by bytes, the array is referred as `cData[].`

Command records and the fields in them are all word aligned. The link field is a 32 bit word. The name field has a length byte followed by variable length name string, null-filled to the

word boundary. In a primitive command, the code field contains byte code, and is null-filled to word boundary. In a compound command, the code field is a 32 bit word, containing the `call,` byte code. The parameter field contains a token list. All tokens are 32 bit words.

This dictionary design was copied from eP32, which was a 32 bit microcontroller. There I used 6 bit machine instructions, and a 32 bit word contained up to 5 machine instructions. In one of the earlier designs, I used 5 bit machine instruction, and I could pack 6 machine instructions to a word. The assembler was designed so that it could pack as many instructions as a program word could allow. In JFM, I already had 67 machine instructions, and 6-bit fields were not enough for them. For convenience, I just allocate 8 bits for instructions, and give you the possibility of using 256 byte code for machine instructions.

I was not particularly concerned about the numbering of byte code. They were assign consecutive numbers as I coded them. However, there is no reason that the numbering could not follow some preconceived order, like Java Byte Code. In fact, there is no reason that you could not build a Virtual Java Machine with this JFM design.

Socket Programming

As far as WiFi is concerned, I started at ground zero. I had no idea what these terms meant: access point, client side programming, server side programming, etc. I heard of TCP/IP, HTTP, HTTPS, but really did not know what they were good for.

Playing with NodeMCU, I saw this example on Arduino IDE, WebLED.ino, to turn its LED on and off. It was on almost every website tutorial talking about ESP8266:

```
#include <ESP8266WiFi.h>
const char* ssid = "SVFIG";
const char* password = "12345678";
int ledPin = 2; // GPIO2 of ESP8266
WiFiServer server(80);
void setup() {
  Serial.begin(115200);
  pinMode(ledPin, OUTPUT);
  WiFi.begin(ssid, password);
  while (WiFi.status() != WL_CONNECTED) { delay(500); Serial.print("."); }
  server.begin();
}
void loop() {
  WiFiClient client = server.available();
  if (!client) { return; }
  while(!client.available()){ delay(1); }
  String request = client.readStringUntil('\r');
  client.flush();
  int value = HIGH;
  if (request.indexOf("/LED=ON") != -1) {
    digitalWrite(ledPin, LOW); value = LOW; }
  if (request.indexOf("/LED=OFF") != -1){
    digitalWrite(ledPin, HIGH); value = HIGH; }
  client.println("HTTP/1.1 200 OK");
  client.println("Content-Type: text/html");
```

```
  client.println(""); //  do not forget this one
  client.println("<!DOCTYPE HTML>");
  client.println("<html>");
  client.print("Led pin is now: ");
  if(value == HIGH) { client.print("On"); }
  else { client.print("Off"); }
  client.println("<br><br>");
  client.println("Click <a href=\"/LED=ON\">here</a> turn the LED on pin 2
ON<br>");
  client.println("Click <a href=\"/LED=OFF\">here</a> turn the LED on pin 2
OFF<br>");
  client.println("</html>");
  delay(1);
}
```

It created a Web client by opening a http webpage with two buttons LED=ON, and LED=OFF. When you click one of the buttons, the client sent back a messages with either ‘LED=ON’ or ‘LED=OFF’ string embedded. The server examined the message and turned the LED on or off accordingly.

I was fairly confused by this example. Does my server have to manage a client to communicate with myself? I like to send an arbitrary message or command to my server and order it to do something I want to do. I didn’t even know what server and client were. But, I knew what I wanted. I liked to have a WiFi network to replace the serial cable to send commands to my computer, and receive responses from it.

I googled WiFi, and checked out all the WiFi books for dummies, and kept myself confused for some months. Then I saw Auduino had another example wifiSoftAP.ino and tried it. It showed that you could turned NodeMCU Kit into a Soft Access Point. It meant that you could build a local network with NodeMCU. That was interesting.

Amid random searches on Google, I hit a pdf book on Socket Programming from IBM, of all the companies. Sockets made lots of sense, and much of the fog and clouds started to lift. After I learnt how to configure sockets for UDP protocol, all my problems were solved. UDP was all I needed. Never mind TCP.

UDP is all I need, because JFM receives commands in packets, and it sends out responses after commands are executed. If there were errors in transmission, JFM would let me know.

Everything worked out fine, until we went to Maker Faire. Things worked while we set up the benches and workstations. When the crowd moved in, many students just could not turn the LED on and off over WiFi, because network traffic was so intense, even we had our own local router. We had to give away kits, when students demonstrated that they had controlled the LED through serial cable.

Serial Monitor and UDP Packets

To communication with JFM over WiFi, I substantially modified the serial IO design in JFM. Original eForth Model assumed a serial IO system sending and receiving ASCII characters.

However, WiFi communication generally assumes sending and receiving packets, a sequence of characters. To make JFM receiving packets, The IO commands are actually significantly simplified. Instead of relying on KEY and ?KEY to receive characters, I use ACCEPT to receive packets, and all the input commands below ACCEPT are eliminated.

eForth provides a line editor so you can edit your input line by backing up and erasing mistyped characters. In WiFi, a client sends packets of characters, and the client always gives you the opportunity to edit the packet before you send it out.

If I could receive a packet directly into the Terminal Input Buffer, then ACCEPT would simple wait for the arrival of a packet and return with the number of characters received.

Where should I place the Terminal Input Buffer? It seems that the best place is the beginner of data[] array. Forth historical reasons, I leave 512 bytes empty at the beginning of the dictionary. Many microcontrollers use this space for reset and interrupt vectors. When data[] is used as a byte array, it is referenced as cData[].

ACCEPT waits for the serial monitor or the UDP receiver to send a packet of characters. If either gets a packets, ACCEPT returns with a character count. Then the text interpreter scans the characters and interprets them.

accept (b u1 -- b u2) Accept u1 characters to b. u2 returned is the actual count of characters received.

```
void accep()
/* UDP accept */
{ while (Udp.parsePacket()==0 && Serial.available()==0) {};
  int len;
  while (!Udp.available()==0) {
    len = Udp.read(cData, top); }
  while (!Serial.available()==0) {
    len = Serial.readBytes(cData, top); }
  if (len > 0) {
    cData[len] = 0; }
  top = len;
  }
```

On the transmitter side, EMIT send a character to the serial terminal and the UDP transmitter. However, the UDP transmitter only adds the character to its output buffer. The whole output packet is only transmitted when JFM executes CR command. It does not make sense to ship each character out in a separate UDP packet. The transmitter involves the following commands:

sendPacket(--) Send an UDP packet out to WiFi network. It is executed only by CR command.

```
CODE sendPacket sendPacket, next,
void sendPacket(void)
{  Udp.endPacket();
   Udp.beginPacket(Udp.remoteIP(), Udp.remotePort()); }
```

`CR` outputs a carriage-return and a line-feed. Prior output characters are accumulated in a UDP packet buffer. This packet is sent out by `sendPacket`.

```
:: CR ( -- ) ( =CR )
  0A LIT 0D LIT EMIT EMIT sendPacket ;;
```