MuP21
Machine Forth

Tutorial #1

MuP21 or P21's real name is M myu P 21
multiple micro processor w/ 3 micros
CPU
Composite Video Analog output coprocessor
Memory interface coprocessor
with a 21 bit CPU w
21 bit address bus and 20 bit data bus

Machine Forth was the successor to Charles Moore's cmForth and OK system
MuP21 was designed by Mr. Charles Moore who named its machine language

What MuP21 Machine Forth is:

It is similar to traditional Forth
It is a new updated Forth Virtual Machine
It is a native code optimizing Forth compiler
The (original) was a target compiler
It is simple

the VM is designed for a simple compiler
the target is designed for 5 bit opcodes
it is small it comiles to 352 bytes original version in FPC
it compiles to 100 words decimal on P21

It is good training for F21 or F32
Mr. Moore's idea of Forth 10 years ago

What Machine Forth is NOT:

It is NOT a magic bullet
It is NOT one implementation,
  8, 16, 20, 24, 32 target and stand alone
It is NOT new - more than 12 years old almost as old as ANS
It is NOT a small subset of some stripped down Forth kernel
  (like Dr. Ting's eForth 1.0)

Don't make the "Lines of Code error:"

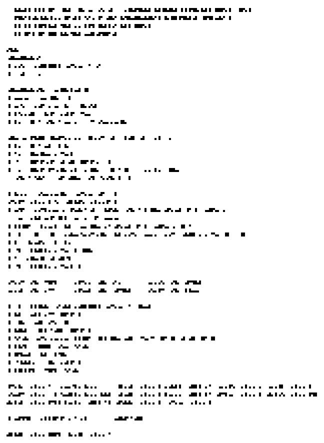
Part of what makes it powerful is that it takes little more than a line of code to define almost all the traditional Forth primitives:

```
00000 JMP jump 0300C JMP T=0 33D4F in @ 3FD7F in ! C0E83 in 2* C328C in 2/ FFFFF in drop F33CC in dup
F3FCF in over FC3F0 in >r F03C0 in r> FF3FC in nop CC2B0 in xor CCEB3 in and CFEBF in + 00C03 in ; 0300C
JMP until C0280 in invert 0C030 JMP (:)
```

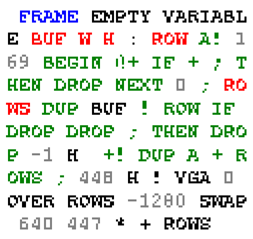
Because they are little more than bits in the hardware
Adding some new things is as easy

```
03C0F JMP -until 03C0F JMP C=0 30D43 in @+ 3CD73 in !+ C3E8F in +* FCFF3 in a! F0FC3 in a
```

The first third of the ANS core fit in a few lines of P21 Machine Forth source



Chuck has moved on to **Color Forth** which is built on his Machine Forth on Pentium.



```
 FRAME EMPTY VARIABL
E BUF W H : ROW A! 1
69 BEGIN @+ IF + ; T
HEN DROP NEXT 0 ; RO
WS DUP BUF ! ROW IF
DROP DROP ; THEN DRO
P -1 H  +! DUP A + R
OWS ; 448 H ! VGA 0
OVER ROWS -1280 SWAP
 640 447 * + ROWS
```

Chuck Moore's **Color Forth** and Jeff Fox's **aha** system are newer and have more new ideas reducing or eliminating dictionary searches at compile time.

To understand them begin with P21 and Machine Forth and then F21 etc.

This presention will have the most brief introduction to Forth itself. It will help if you are familiar with Forth. BUT If you are familar with Forth it can be a big problem. You need to keep an open mind, have fresh eyes.

The first thing I did was spend a few years pounding a square ANS Forth peg into a round Machine Forth hole. I learned something. It took me a years to begin to appreciate the page of Machine Forth that Chuck had written. Jeff Fox

Forth the Language

Parameter stack, return stack
Words and spaces almost no syntax like objects
Factored simplicity to solve problems
Extensibility, tool to create 4GL application specific language extensions to Forth

Competes with LISP for abstraction
Competes with C or assembler for closeness to metal

```
: NEW-WORD ( -- )
 DO-SOMETHING AND-SOMETHING-ELSE ;

: DEMO ( n -- ) TAKE-PARAM-INIT NEW-WORD FINISH-UP ;

  manipulate the parameter stack with RPN

: MEM-AND ( a n -- )  \ and a number with the contents of a location in memory
and store it back
 OVER ( a n a )         ( n1 n2 -- n1 n2 n1 )
 @ ( a n1 n2 )          ( a -- n )
 AND ( a n1-and-n2 )    ( n1 n2 -- n1-and-n2 )
 SWAP ( n a )           ( n1 n2 -- n2 n1 )
 ! ( -- )               ( n a -- )
; \ return
```

Q. Are WORDS just subroutines?
A. Some Forth WORDS are subroutines, they should be SHORT subroutines
Many words are like Macros
Many words are named data
Many words are like objects with code or data

A Forth compiler is a one pass compiler with a name and code dictionary that parses source code and generates either native code or code for a Virtual Machine

Part 2

(sections will be added as the videos are shown)

What is Machine Forth...

Begin with understanding the MuP21 machine and the new Virtual Machine

The P21 has on chip stacks a total of 13 registers:

 6 data stack cell registers
 4 return stack cell registers
 1 PC program counter
 1 A addressing register
 1 control register with
  1 bit to turn video on/off
  2 bits to select a ROM page (4 * 256KB)

Before P21 or F21 ran I ran about 30 Forth systems on variations of the S21 simulator, benchmarked them and published the result in a paper at FORML.

```
  Classic or ANS Forth ideas and models

      ITC     IP=A SP=T W=S RP=R UP=R1

      DTC     IP=A SP=T RP=R UP=S       faster  P21Forth

      BTC     IP=A SP=T RP=R UP=S       fast small

      STC     IP=PC T+5 R+3S UP=mem     slower bigger because of 10 bit pages
                                         page calls = 15 bits, 2 words

      Native  IP=PC T+5 R+3S UP=mem     fastest, very compact 5 bit opcodes
                                        designed for simple native code
```

P21 supports three address spaces:

1 Megaword of 20 bit wide DRAM
1 Megabyte of 8 bit wide ROM/SRAM/FLASH as 4 256 pages
1 Kiloword of 20 bit wide SRAM decoded

All Offete boards memory map I/O devices into the SRAM address space and have no SRAM space. The first board used a PCMCIA SRAM card to boot so the select line was called SRAM but generally means 8 bit boot ROM. The P21 has a control line called I/O that is really the 20 bit wide SRAM/ROM/FLASH address space. These same lines are called SRAM and I/O on P21 are called ROM and SRAM on F21.

Memory bus width is 20 bits, regsiter width is 21 bits:

20 bit numbers and addresses FFFFF 12345
21 bit numbers and addresses 140000 100000 1C0000

numbers and patterns in a cross compiler

on one machine
5V 5V 5V 5V = true true true true
on another machine
5V 5V 5V 5V = false false false false

to convert between Intel and Motorola
you must XOR data with FFh

on Chuck's MISC machines
5V 5V 5V 5V = false true false true

to convert between MISC and PC
you must XOR data and addresses with AAAAA

numbers and patterns in coprocessors

the video coprocessor instructions are patterns, they must be xored with AAAAA to be logically manipulated by the CPU.

Chuck's code always has a certain look, his traditional Forth, Machine Forth, and Color Forth are very similar. Short words, few comments. His style has become more consistent with shorter words.

Let's go back about ten years...

60 words Chuck's first MuP21 Machine Forth implementation:

```
 Glossary to all source words      (for those interested in how it works)

  H ( -- a )    \ code dictionary pointer
  LOC ( -- a ) \ name a location
  Hi ( -- a )   \ instruction opcode pointer 0,4,8,12,16 for slots 0 1 2 3 overflow
  Hw ( -- a )   \ instruction word pointer
  ALIGN         \ to start of word
  ORG ( a -- ) \ set compile address
  SWITCH ( a1 -- a2 ) \ switch compile address
  IS ( -- a ) \ name the next executable location
  mask ( -- a ) \ address of array of masks for instructions in slots 0 1 2 3
  p, ( p -- )   \ compile a physical pattern to memory
  #, ( n -- )   \ compile a number or address to memory
  ,w ( n -- )   \ compile a word for target memory location to target ROM
  ,i ( n -- )   \ compile an instruction and write to target ROM
  INST ( n -- ) \ defines a named instruction opcode
  com ( n -- _n ) \ -1 xor invert opcode
  nop           \ nop opcode
  JMP ( a -- ) \ define branching instruction
  begin ( -- a ) \ begin macro
  -;            \ semicolon with auto recursion
  p ( p -- )    \ compile physcial pattern literal macro
  -p ( p -- )   \ compile physcial pattern as inverted literal macro
  # ( n -- )    \ compile number as literal opcode
  -# ( n -- )   \ compile number as inverted literal macro
  jump ( a -- ) \ compile unconditional paged jump opcode
  T=0  ( a -- ) \ compile if T=0 paged jump opcode
  C=0  ( a -- ) \ compile if carry=0 paged jump opcode
  call ( a -- ) \ compile paged call opcode
  until         \ until T=true macro
  -until        \ until carry=true
  ':            \ : defining word
  if ( f -- f ) \ if then macro
  -if           \ if carry macro
  skip ( -- a ) \ unresolved branch compile
  then          \ if then macro
  else          \  if else then macro
  while ( f -- f ) \ begin while t=true macro
  -while        \ begin while carry=true macro
  repeat        \ begin while/until repeat macro
  @+ ( -- n )   \ opcode fetch from A address, increment A
  !+ ( n -- )   \ opcode store using A address, increment A
  ! ( n -- )    \ opcode store using A address
  2* ( n -- 2n ) \ opcode 21 bit 2*
  2/ ( n -- n/2 ) \ opcode 20 bit 2/ Carry and Sign unchanged
  +* ( n1 n2 -- n1 n2 | n1 n1+n2 ) \ multiply step opcode
     \ conditional nondestructive add opcode DUP 1 AND IF OVER + THEN
  xor ( n1 n2 -- n1_or_n2 ) \ opcode
  and ( n1 n2 -- n1_and_n2 ) \  opcode
  + ( n1 n2 -- n1+n2 ) \ opcode
  r> ( -- n ) ( R: n -- ) \ opcode
  a ( -- a )    \ opcode
  dup ( n -- n n ) \ opcode
  over ( n1 n2 -- n1 n2 n1 ) \ opcode
  >r ( n -- ) ( R:  -- n ) \ opcode
  a! ( a -- )   \ move T to A register opcode
  ;'            \ return opcode
  ljump ( -- ) \ compile long jump macro to  word
  drop ( n -- ) \ opcode
  @ ( -- n )    \ opcode fetch from A address
```

some macros in P21 Machine Forth

Forth	P21 Machine Forth Macro
@	a! @ (@a)
!	a! ! (!a)
SWAP	a! >r a r>
	over >r >r drop r> r>
ROT	>r >r a! r> r> a
OR	over com and xor
0	dup dup xor
-1 (21-bit)	dup dup com xor

I prefer the opcode names @A and !A for what Chuck called @ and ! in Machine Forth. This allows @ and ! to be the above macros and operate like traditional Forth.

The splitting of the atomic nature of the Forth @ and ! words provides Machine Forth with:

Addresses used for @ and ! are cached in a register this keeping a copy on the stack and avoids the stack juggling that that requies.

@A+ and !A+ auto-increment opcodes provide a pointer or loop counter increment with no additional time compared to a @A or !A non-incrementing instruction.

The A register also can be used like a fast local variable when it is not being used as a pointer.

The A register also can be used like a 1 cell deep third stack or a global variable to pass an agrument through routines

Part 3

Control Flow:

like ANS Forth you may mix and match
IF ELSE THEN BEGIN WHILE UNTIL REPEAT

even more with P21 Machine Forth
-IF -WHILE -UNTIL

Traditional Forth

IF WHILE UNTIL drop the flag

P21 Machine Forth

no drop like **DUP IF**

Note about eForth:

eForth came out about the same time Chuck started working on MISC chips. Both Dr. Ting and I implemented eForth on MISC chips. MISC chips have about the same number of opcodes as eForth has CODE words.

Ting removed the meta-compiler, the ANS compatibility layer, VARIABLE CONSTANT and DOES> from the kernal, converted it from Forth to MASM source format, and distributed dozens of version in this form. As a result...

Bill's eForth 1.0 DO LOOP in ANS part FOR NEXT in core
Bill's eForth 2.0 DO LOOP in ANS part BEGINs only in core

What Makes Machine Forth different than Traditional Forth?
Chuck has said that as a result of the changes that a completely different style of programming drops out.
Why?

One change was to keep the number of requirments down to a minium. Many of the words in traditional Forth were no longer needed. Chuck didn't throw these words out because he didn't know to implement them. He threw them out because he replaced them with things that were simpler and better.

Another change was the minimal stack depths. At the time he did this common advice was that Forth needed only a few hundred stack cells. Chuck said that P21 was designed to run the VLSI CAD software that he had been writing and using and that his software only needed a few stack cells, that good Forth only needed a few stack cells and should not waste them.

He had gone from DO LOOPs to FOR NEXT in cmForth for the Novix because it was simpler and closer match to how the software worked. Bill's eForth 1.0 used FOR NEXT in the kernel and added DO LOOP stuff into the ANS layer.

Bill threw out FOR NEXT from the kernel in version 2.0 and replaced it with only BEGIN constructs in order to make it smaller, faster and easier to read. At the same time Chuck Moore had rejected both FOR NEXT and DO LOOP saying that DO LOOPs came from and belonged in FORTRAN not in Forth.

Chuck explained that DO LOOP was often used to increment a loop or manipulate pointers and that it was a clumsy, resource wasteful, and slow way to do it compared to BEGIN and that using a third as many different words would make the source code cleaner and clearer. That is what I saw in Bill's changes in eForth from 1.0 to 2.0 and I saw it again in P21 Machine Forth.

With only a 4 cell deep hardware return stack on P21 the idea of nested DO LOOPs and nested Forth definitions is problematic. DO LOOPs do look like Fortran on MuP21.

Comparing Traditional Forth and Machine Forth looping
Traditional Forth DO LOOP uses two return stack cells

```
\ store 0 to N in character cells from a1 to a2

0 a2 a1 DO DUP I C! 1+ LOOP DROP
a1 a2 OVER DO I OVER - I C! LOOP DROP
a1 a2-a1 0 DO I OVER + I SWAP C! LOOP DROP
```

```
0 a1
BEGIN
  OVER OVER C!
  CHAR+ SWAP 1+ SWAP
  DUP a2 =
UNTIL DROP DROP
```

P21 Machine Forth BEGIN version w/ no return stack cells

```
a1 a! 0 DUP
BEGIN
   DROP DUP !+ 1+
   A a2 XOR
WHILE REPEAT DROP
```

The Machine Forth BEGIN version even with the extra DUP and DROP is shorter than the traditional Forth version. The first thing that it does is store a1 into the Address register. The auto-increment instruction eliminates quite a bit of the traditional code.

Chuck said other people spend of lot of transistors on memory protection and hardware fault detection to catch stack overflow errors. They must also make the software more complex to deal with this hardware. They spend a lot for that. They spend more on a software layer to trap stack overflow and underflow bugs.

Chuck said that other people spend a lot of transistors on slow stacks using stack pointers into memory to be able to treat stacks as arrays and build addressable stack frames. They do this because in languages that don't have a parameter stack they must do this on the return stack. They spend a lot for the hardware and they spend a lot at runtime on the software to use this mixed stack array idea.

Forth uses two stacks, parameter and return stacks so that one doesn't have to mix addressable arrays and stack frames with actual stacks. Forth uses real stacks Chuck built real stacks in hardware, fast, on chip register stacks. He has no stack pointer, no addressing into stacks, no stack frames. Forth should treat a stack as a stack since that was the idea from the beginning and what works nicely when you have more than one stack.

Some old Forth words to be avoided. As always. Because you have a real parameter stack, ie. compsci def a stack not a mix of stacks and arrays, there is no stack pointer in Machine Forth. Stacks are in on chip stack registers in real stacks. Therefor the old Forth word DEPTH isn't applicable. In fact implemening it in one ANS Forth port direclty to Machine took fit everything but DEPTH in eForth 1.0 in 1K words and DEPTH took much of another 1K.

DEPTH is a sign that a program that has lost control, it has lost count of what is on the stack so it is already out of control.

"Stay out of the DEPTH."

PICK assumes that the stack is not a stack but an ARRAY a real stack does not allow PICK. You shouldn't need it.

"Don't pick you code in public."

ROLL also assumes that the stack is really an array.

"This is low fat computing, no ROLLs or Donuts."

CS-PICK is even worse ANS Forth is a 4 stack virtual machine
parameter, return, control, floating point
CS-ROLL is the same. Machine Forth is a two stack machine
parameter, return.

These words are essential to write C style code in Forth. They are not what Chuck calls Forth. He was very happy to "solve the stack overflow and underflow problem once and for all" with a simple solution and make stacks that work like Forth stacks and not at all like C stacks.

This made them very cheap and very fast and the software very simple.

another innovation:

Circular Stacks.

```
T  S S2 S3 S4 S5
     ^_________^

   R R1 R2 S3
   ^________^
```

protect stack overflow or underflow from damaging memory contents

can be used to store repeating pattern
no dup or fetchs are needed to write any numbers of copies

Circular parts of the stacks can be spun backwards
R> R> R> R> leaves the return stack unchanged on MuP21

(Not on the Video)

One final note on control flow is the technique that Chuck has been using in Color Forth that also can be used in many Machine Forths, of auto-recursion using his tail-recursion function. In some versions of Machine Forth **;** will automatically perform the tail-recursion function of **-;** when legal, that is when the preceeding word is a call and no **THEN**s have been executed. (**-;** is an optimization of **;** where instead of a call to a word that returns to a return, instead of compiling a return to follow a called word, it converts that call into a jump.)

So if a system does not use the Forth word **SMUDGE** it can end a word with a recursive call to itself followed by **-;** which then removes the return stack overhead of recursion and makes something equivalent to **BEGIN AGAIN** but with only one word. As in:

```
: MYWORD ... ... IF ; THEN ... ... MYWORD -;
```

the same as:

```
: MYWORD BEGIN ... ... IF ; THEN ... ... AGAIN ;
```

except shorter and without the uneeded **;** at the end.

Or one of Chuck Moore's examples of Color Forth from 1x Forth:

```
WORD~~~ IF ~~~ WORD ; THEN ~~~ ;
```

Part 4

Timing is easy to calculate:

This makes it easy to compare the speed of different coding options. The S21 simulator makes it even easier.

Instruction timing: All opcodes take 10ns

Memory setup time is 25ns which make cycle times
40ns for high speed SRAM, ROM, FLASH 8 or 20 bit
50ns for onpage DRAM
150ns for offpage DRAM
250ns for slow speed SRAM, ROM, FLASH 8 or 20 bit

SRAM/ROM/FLASH 8 and 20 bit memory spaces are mapped twice, one with 40ns timing, one with 250ns timing.

Stack operations with no branching run at 10ns each in SRAM.

```
Memory prefetch starts after the last memory access
instruction executes

SRAM: 100 mips max stack opcodes only

DROP DUP A! >R  \ start prefetch 40ns instructions
                \ 40ns instruction fetch 40ns total
@ DUP DUP DUP   \ 10 ns @
                \ +40 ns data fetch
                \ begin prefetch 30ns more instructions
                \ 10ns of instruction fetch after last DUP 90ns
A! DROP OVER A  \ 40ns instructions and prefetch

12 instructions 170ns total ~70 mips
```

```
Onpage data DRAM: 80 mips max stack opcodes only

DROP DUP A! >R  \ start prefetch 40ns instructions
                \ 50ns instruction fetch 50ns total
@ DUP DUP DUP   \ 10 ns @
                \ +50 ns data fetch
                \ begin prefetch 30ns more instructions
                \ 20ns of instruction fetch after last DUP 110ns
A! DROP OVER A  \ 50ns instructions and prefetch

12 instructions 210ns total ~60 mips
```

```
Offpage data DRAM: 80 mips max stack opcodes only

DROP DUP A! >R  \ start prefetch 40ns instructions
                \ 50ns instruction fetch 50ns total
@ DUP DUP DUP   \ 10 ns @
                \ +150 ns data fetch
                \ begin prefetch 30ns more instructions
                \ 120ns of instruction fetch after last DUP 310ns
```

```
A! DROP OVER A  \ 50ns instructions and prefetch

12 instructions 410ns total ~30 mips



Ripple Carry on + and +*  8 bits/10ns

the time requred for addition is defined by the
nubmer of bits through which carry must move

doing an add with these numbers already on the stack

7F 1 +          \ 10 ns
7F000 1000 +    \ 10 ns
7FFF 1 +        \ 20 ns
FFFFF 1 +       \ 25 ns

Putting + or +* in slot 0 will always work.

 sometimes NOPs are needed BEFORE + or +*

 NOP 7FFF # A! NOP
 1 # A NOP +      \ 20 ns ok now that NOP + gives 20ns after A changes T

 NOP 7FFFF # A! NOP
 1 # A +          \ 25 ns ERROR only 10ns after A changes T

 NOP 7FFF # A! NOP
 1 # A NOP NOP
 +                \ 25 ns ok now that NOP + gives 30ns after A changes T
```

Q: Why before?
A: This exposes one of P21's internal hardware innovations.

All 25 Mup21 opcodes are executed on every cycle!

In RISC there are bits that designate which register to use MISC is a zero operand architecture meaning implied top of stack or hard wired registers in each instruction

In RISC:

> fetch an instruction
> decode the instruction bits and register select bits
> setup the paths between the ALU and the proper registers
> execute the ALU operation

In MISC:

fetch the instruction
execute all opcodes and decode one to select
select the output of the one decoded

So internally MISC has faster register access than RISC or CISC.

At intel at the same geometry with cache the instructions take 120ns with pipelining if it doesn't stall they can get 40ns instructions MISC is getting 10ns instructions in this geometry w/o cache or pipelines and with nearly 1000x more transistors, cost and power consumption and of course much more complex programming.

```
20 bit vs 21 bit math

-1 #
 is a 20 bit literal ie FFFFF (no carry)

DUP DUP COM XOR
 is a 21 bit literal ie 1FFFFF (carry set)



Using +* as a fast loop counter:

+* is equivalent to   DUP 1 AND IF OVER + THEN

BEGIN +* UNTIL   or
BEGIN +* -UNTIL  type constructs take advantage
of the nondestructive nature of +* because of its
OVER thus

BEGIN DUP 1 AND IF OVER + THEN -UNTIL
compiles to one word of memory and runs very quickly.



Using +* as a multiply step:


a 4-bit x 4-bit multiply of N1 N2

: 4x4
>r 2* 2* 2*
2* r> nop nop \ shift N1 left 8 times
+* 2/ nop nop
+* 2/ nop nop
+* 2/ nop nop
+* 2/ >r drop \ 4 conditional adds and shifts
r> ;



1010 0011 4x4 \ example
```

```
10100000 0011 \ after 4 shifts
10100000 10100011 \ after 1st conditional add
10100000 1010001  \ after 1st shift
10100000 11110001 \ after 2nd +*
10100000 1111000  \ after 2nd shift
10100000 111100   \ after 3rd
10100000 11110    \ after 4th
11110             \ result
```

fastest when T is the has fewer bits
Chuck uses multiplies with 12 and 16 bit results in OKAD
10x10 max
smaller multipies are faster
many compilers do something similar ven on Pentium

Machine Forth literals:

Chuck had explict ones: # opcode is like LIT
Some machine Forth comilers have had auto-literals in
compile mode like traditional Forths.

Machine Forth is not Fortan

problem: given X define Y to perform 10 X

Fortran solution:

```
\ after we define something like

': (DO)          \ machine Forth DO LOOP primitive
 r> a!           \ save (DO) return address
 >r >r           \ 1 0 becomes 0 1 on r stack
 a >r ;          \ return

: DO ( -- a )
 ' (DO) # compile,   \ compile a call to (DO)
 dp @ ;              \ return address
 IMMEDIATE

: (LOOP)
 a r> a! r>          \ save A, get (loop) return and save, get loop termination
 r> over over xor    \ get I, make two copies, compare
 IF drop 1 # nop nop \ increment I, return address in A
   + >r >r @a        \ get jump address from return address location
   >r a! ;           \ restore A, branch back to A
 THEN drop drop      \ drop flag, drop loop count
 drop @a+            \ drop loop termination, fetch branch address increment
pointer
 drop a push a!      \ restore a, return to incremented return pointer
 ;

: LOOP ( a -- )
 ' (LOOP) # compile,    \ compile (LOOP)
 compile,               \ compile loop address
 ; IMMEDIATE
```

```
\ now the Fortran programmer can say:

: Y 10 # 0 DO X LOOP ;
                      \ 20 characters in soure
                      \ 4 return stack locations and A
                      \ 300ns loop overhead for each X
\ compiles
\ 10 # dup dup xor    \ fast 0
\ (DO)                \ 23 word overhead
\ X                   \ 6 words here
\ (LOOP)              \ 300ns overhead on each loop
\ X-adr
\ ;                   \ 29 words total 3000ns overhead



\ With a short loop better to write:

: Y X X X X X X X X X X ; \ 24 chars source
                          \ 2 return stack locations
                          \ no overhead for each X
\ things like

: Z 4 0 DO 3 0 DO X LOOP LOOP ;

\ in this case it is much better to write:

: Y X X X ; : Z Y Y Y Y ;



\ looping techniqes

\ shift until carry

 1 BEGIN ... 2* -UNTIL DROP         \ 1 to 20 shifts

\ multipy step till carry

2 1 BEGIN ... +* -UNTIL DROP DROP   \ 1 to 1/2 meg

\ add till carry

1 0 BEGIN ... OVER + -UNTIL DROP DROP \ 1 to 1M

\ add till zero

1 0 BEGIN ... OVER + 0= UNTIL DROP DROP \ 1 to 1M
1 0 BEGIN ... OVER + WHILE REPEAT DROP DROP \ 1 to 1M

\ typical Machine Forth
\ pointer equals end of loop

a1 a!
DUP BEGIN DROP ... @A+ ...
       a a2 # xor WHILE REPEAT DROP  \ adr based

\ a! first thing
\ no need to clean up A at end
```

Part 5

Original Machine Forth Source
Chuck Moore's Cross Compiler for FPC
from OK101f Source in P21Forth.

```
\ ok16a.seq cm, cht, jf 93,94,95   Original Machine Forth for MuP21  1993
\ written in FPC where the 16 bit implementation requires doubles to
\ cross compile the 20 bit target for MuP21
\ r! and r@ are in the ROM burner

HEX
VARIABLE H
: LOC   CONSTANT  DOES> @   H ! ;
: 2,    , , ;

VARIABLE Hi   VARIABLE Hw
: ALIGN   10 Hi ! ;
: ORG   DUP . CR H !  ALIGN ;
: SWITCH   H @  SWAP  ORG ;
: IS   H @  Hi @ 10 / +  0 2CONSTANT ;

CREATE mask  AA800. 2,  55400. 2,  32A. 2,  D5. 2,
: p,   H @ R!  1 H +! ;
: #,   AAAAA. 2-OR p, ;
: ,w   Hw @ R@  2-OR  Hw @ R! ;
: ,I   Hi @ 10 AND IF  0 Hi !  H @ Hw !  0. p,  THEN
   Hi @ mask + 2@ 2AND  ,w  4 Hi +! ;

: INST   2CONSTANT   DOES> 2@  ,I ;
C0280. INST com   FF3FC. INST nop
: JMP   2CONSTANT  DOES> 2@  BEGIN  Hi @ 8 AND WHILE  nop  REPEAT
  ,I  3FF AND 155 -OR 0 ,w  ALIGN ;
: begin   BEGIN  Hi @ 10 AND 0= WHILE  nop  REPEAT  H @ ;
: -;'   Hw @ R@  OVER 4000 AND  IF 4000  ELSE 8000  THEN 0 2-OR  Hw @ R! ;
: p   3314C. ,I  p, ;
: -p   FFFFF. 2-OR  p com ;
: #   AAAAA. 2-OR p ;
: -#   55555. 2-OR p ;

00000. JMP jump   0300C. JMP T=0    03C0F. JMP C=0    0C030. JMP call
                  0300C. JMP until  03C0F. JMP -until

: ':   begin  .head CONSTANT  DOES> @  call ;
: if   155 T=0  Hw @ ;
: -if   155 C=0  Hw @ ;
: skip   155 jump  Hw @ ;
: then   DUP >R >R  begin  3FF AND 155 -OR 0  R> R@ 2-OR  R> R! ;
: else   skip  SWAP then ;
: while   if  SWAP ;
: -while   -if  SWAP ;
: repeat   jump  then ;

30D43. INST @+                      3CD73. INST !+    3FD7F. INST !
                  C0E83. INST 2*    C328C. INST 2/    C3E8F. INST +*
CC2B0. INST -or   CCEB3. INST and   CFEBF. INST +
F03C0. INST r>    F0FC3. INST a     F33CC. INST dup   F3FCF. INST over
FC3F0. INST >r    FCFF3. INST a!
00C03. INST ;'

: ljump ' >body @ 0 #  >r ;' ; \ long jump

FFFFF. INST drop  33D4F. INST @
```

---

page created 01/11/01 Jeff Fox