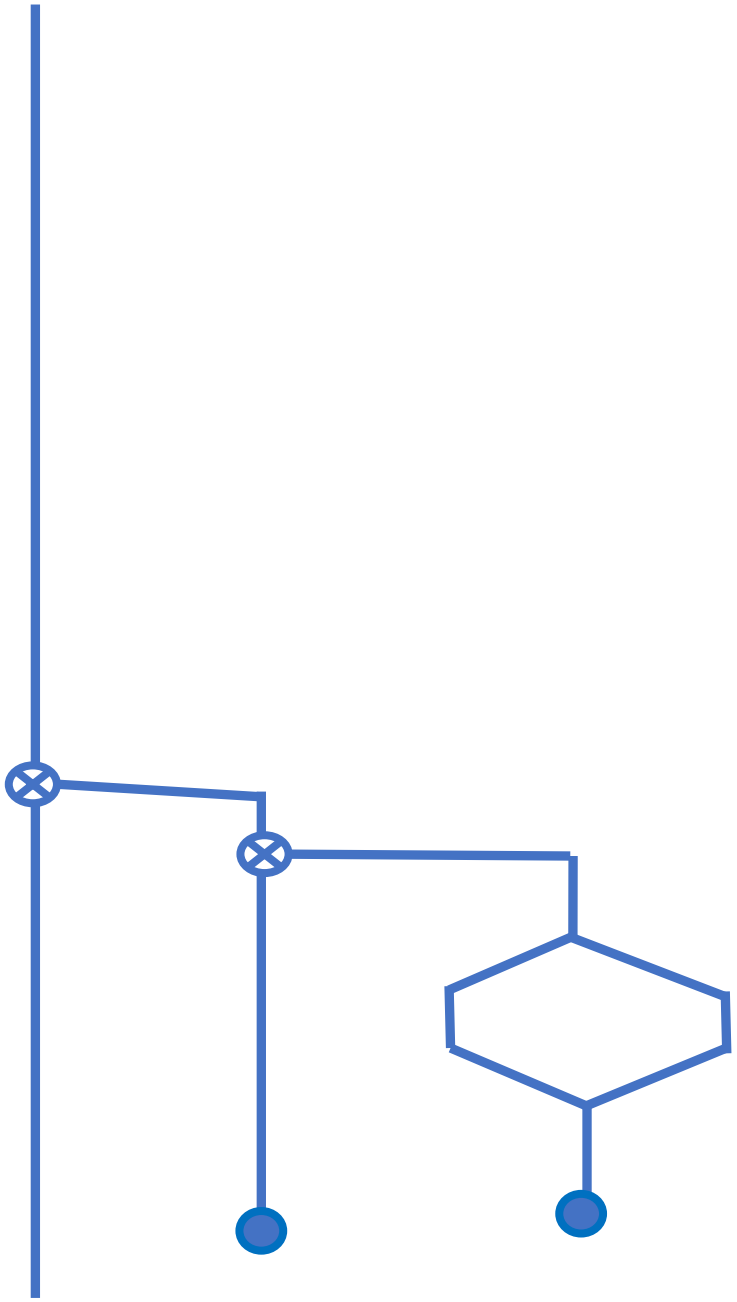


Forth With Small Silicon

SVFIG
Nov. 16, 2024
Bill Ragsdale

The Premise

- Imagine our time is limited for silicon development.
- If execution speed and memory don't matter . . .
- How many CPU opcodes are enough?
- This is an exploration. Is it practical?

Execution Levels

- Silicon level, 14 actions; all handling 32 bits.
  - All through the stack; no register-to-register transfers.
- Code-level, 14 op-code assembler.
  - Linear, no nesting, no looping mechanism.
  - Can only use silicon level actions.
  - Opcode bits could be equivalent to microcode.
- High-level
  - Colon definitions, nesting, branching, looping.

Registers

SP	stack pointer
RP	return stack pointer
IP	interpretive pointer
UP	user area pointer

Key Memory Areas

Data Stack

Return Stack

User Area

Dictionary

Magic Array

Assembly Language

- Memory
  - @ ! SP@ SP! RP@ RP!
- Math
  - Literal + Magic
- Logic
  - Nand
- Control
  - Exit Execute Branch ?Branch

Note: This is the ENTIRE assembly language! 14 opcodes

Meta Level, to bootstrap

- Dictionary
  - DP, a user variable
- Defining
  - CREATE USER
- Compiling
  - CODE C; : ; COMPILE

Stack Manipulation

Built on: `@` `!` `sp@` `sp!`

Generating

`dup` `drop` `swap` `over` `rot` `nip` `tuck`

Simple Code Examples

```
code dup  ( n1 --- n1 n1 ) \ duplicate top of stack
   SP@  @   c;

code drop ( n1 --- )   \ drop from stack
   SP@  !  c;

code swap ( n1 n2 --- n2 n1 )
  SP@ -12 + !     SP@ -12 + !
  SP@ -20 + @     SP@ -12 + @  c;
```

Some are simple and direct. Some quite complex.

Complex Code Example

```
Complex, uses 32-bit operators to handle 8 bit values.
Could be written at high level as:

: c! ( n1 addr1 --- )  \  store byte n1 at addr1
    dup @ -255 and rot 255 and or swap ! ;

But expands at the silicon level to:
code c!
  sp@ @  @ -255 nand SP@ @ nand
  RP@  -8 + !  RP@  -12 + !  RP@ -16 + !
  RP@ -12 + @  RP@   -8 + @  RP@ -16 + @
  255   nand   SP@ @ nand
  SP@ @ nand   SP@  -8 + !  SP@ @ nand   SP@ -12 + @  nand
  SP@ -12 + !  SP@ -12 + !  SP@ -20 + @  SP@ -12 + @
  !   c;
```

Logic

Built on silicon "nand" using DeMorgan's Theorem.

INVERT is "`dup nand`"
AND is "`nand invert`"
OR is "`invert swap invert nand`"
XOR is very complicated

Logic Examples

```
code invert   sp@  @  nand  c;

code and   nand  sp@  @  nand c;

code or  SP@ @ nand  SP@ -8 + !
         SP@ @ nand  SP@ -12 + @  nand  c;

: xor   dup >r   over nand   swap  invert
        r>  invert   nand  nand  invert ;
```

Math

Built on: + nand

```
code invert     SP@  @  nand  ;

code negate     SP@  @  nand  1 +  ;

code -          SP@  @  nand  1 +  +  c;
```

Code-Level

Built from silicon code words.

IF	ELSE	THEN	BEGIN
AGAIN	WHILE	REPEAT	UNTIL
C,	,	C!	C@
-	R>	>R	R>DROP
1LSHIFT	1RSHIFT	LROLL	RROLL
LSHIFT	RSHIFT	+!	XOR
OR	AND	TUCK	NIP
ROT	SWAP	OVER	DUP
DROP	INVERT		

Bit Manipulation

Generating:

`Rshift, Lshift, 1Rshift, 1Lshift, Rroll, Lroll.`

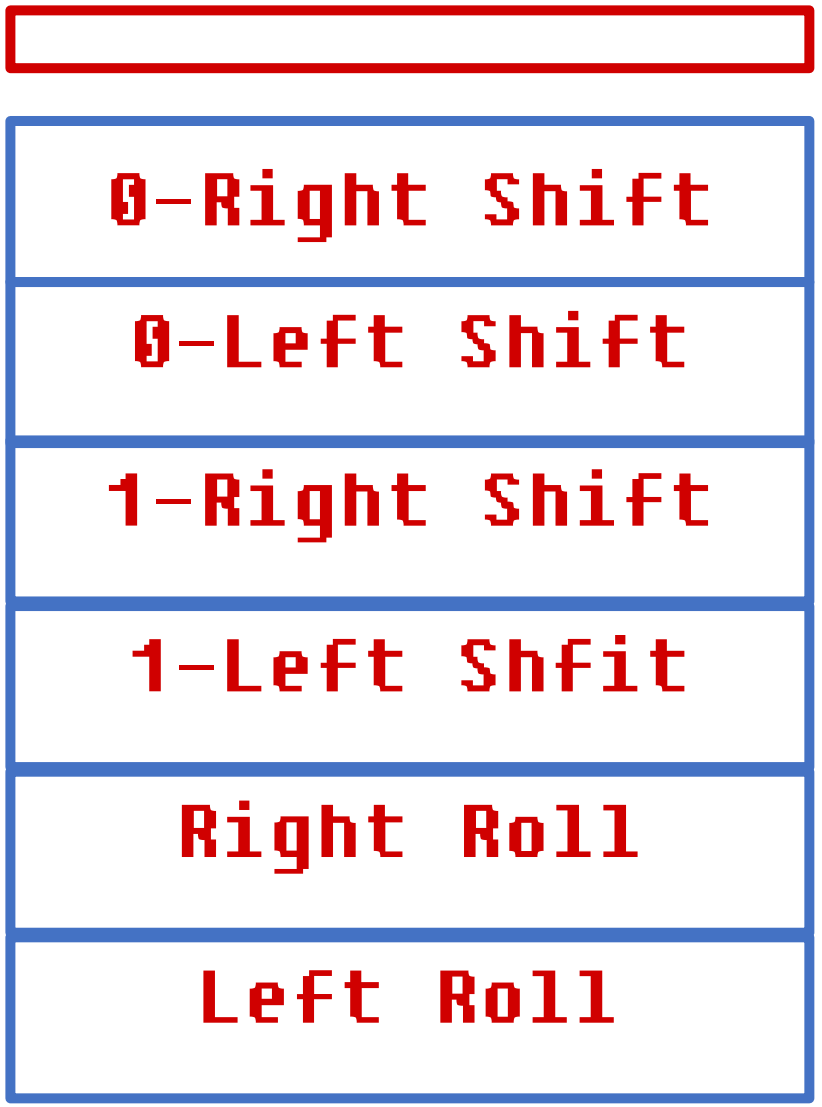
Build on: ‘`magic`’ a ram array of 192 cells.

A 32-cell array for each word.

Write into base memory cell.

Read from a linked, offset cell.

Magic Array



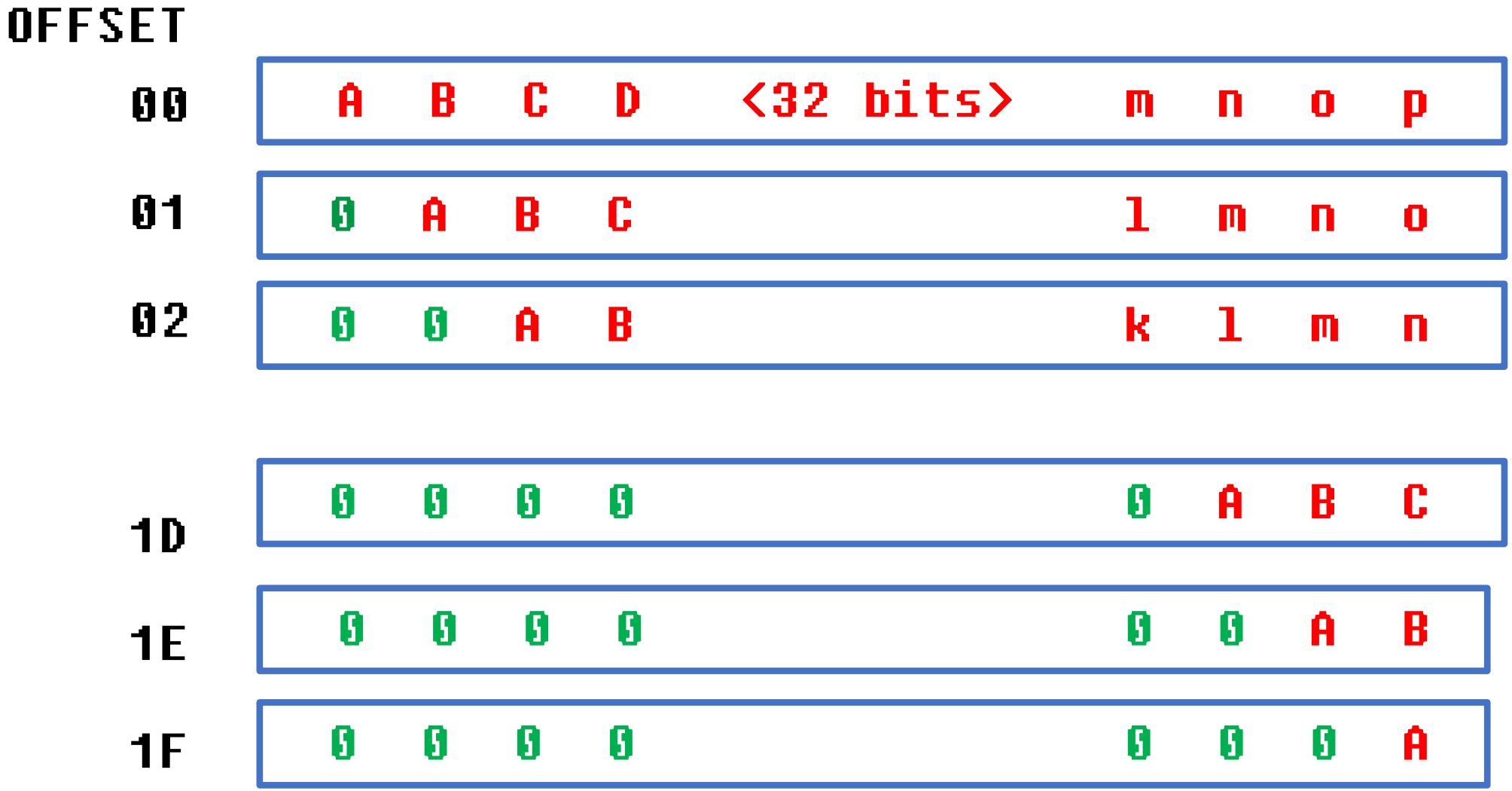
Base Address

Write to the base address.

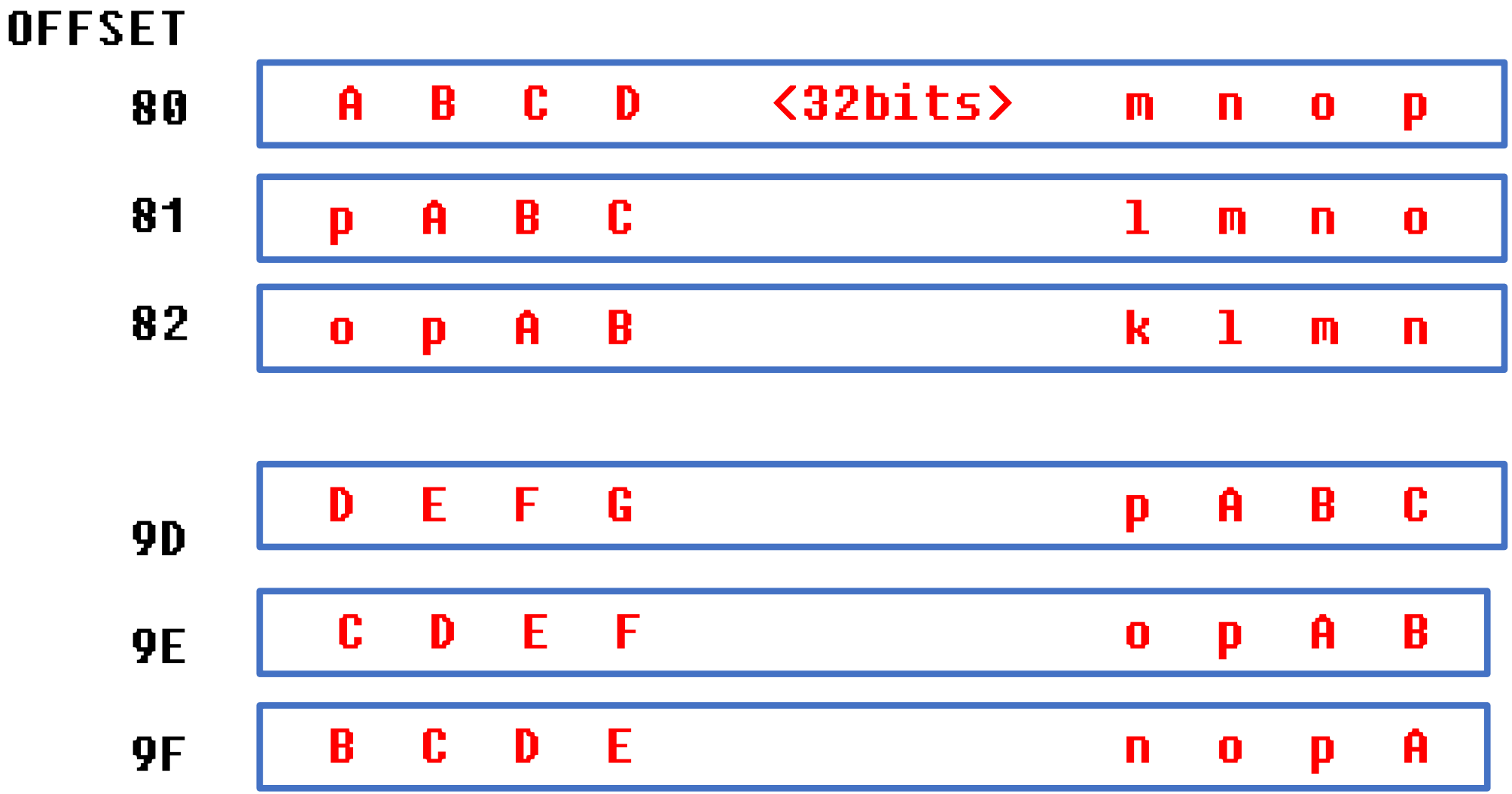
Read adjusted cell at an offset.

A local, 32 bit bus between cells does the shifting.

Right Shift, 32 bits, zero fill



Right Roll, 32 bits



Roll Example

```
: Rroll ( n1 n2  --- n3 )  \ roll n2 bits right
   swap  magic !           \ load register
   0x80 magic +  +  @  ;   \ access with offset

0x0F  4  Rroll h.    See:    F0000000  ok
```

Return Stack Manipulation

```
: >r ( n1 --- ) ( R: --- n1 ) \ move to return stack
   rp@ @      rp@ -4 + !    rp@  !
   rp@ -4 +   rp!   ;

: r> ( R: n1 --- ) ( --- n1 ) \ move from return stack
   rp@ 4 + @  rp@ @     rp@ 4 + !
   rp@ 4 +    rp!   ;
```

Flow Control

Built on: `branch,` `?branch.`

Generating:

```
if  else  then

begin  until

begin  while  repeat

begin  again.
```

Conditionals

```
: HERE          dp @ ;
: >MARK         HERE 0 , ;
: <MARK         HERE ;
: >RESOLVE      HERE 4 +  SWAP ! ;
: <RESOLVE      , ;

: IF     COMPILE ?BRANCH  >MARK                  ; IMMEDIATE
: THEN   COMPILE _THEN >RESOLVE                  ; IMMEDIATE
: ELSE   COMPILE BRANCH >MARK  SWAP >RESOLVE ; IMMEDIATE

: test  1 if  ." one"  else ." not one"  then ;
```

My Simulation

- Silicon primitives are in a W32Forth vocabulary Silicon-level.
- Code words, only using silicon primitives in the vocabulary Code-level.
- Language expanded in the vocabulary High-level.
- Future: multiply and divide.

Implementations

- My simulation follows Win32Forth.
- Actual implementation could be ITC, DTC, subroutine threaded or . . .
- Code words could have high bit set; colon definitions have high bit clear. Allow mixing of high-level and code, inline.
- The ‘magic’ array in ram uses 32-bit cell images of all rolling and shifting possibilities. 192 cells.

Conclusions

- Very stack intensive.
- All operations pass through the top two stack items.
- No arithmetic logic unit. Just an adder and 32-bit nand.
- The bits of silicon opcodes could be microcode selecting control and flow in silicon.
- Could be implemented in discrete integrated circuits.