c18 colorForth Compiler

Chuck Moore

chipchuck@colorforth.com

November 2001

Abstract

c18 is a tiny computer that can be replicated many times on a silicon die. Its architecture and instructions are optimized for Forth. colorForth is a dialect of Forth that features preparsed source code and the use of color for punctuation.

A compiler has been written in colorForth that compiles colorForth source to c18 machine code. This code can be placed into c18 ROM or RAM to be executed by a simulator. Or into serial EPROM/Flash to be executed by a c18, when it exists.

Emphasis here is on how simple a colorForth cross-compiler can be. The c18 is interesting because it can run extremely fast using little power: 2400 Mips @ 20 mW.

c18 instructions

These are the current c18 instructions that must be compiled. As the design evolves changes are obviously easy to make.

	Register
T	Top of stack
S	2nd number on stack
R	Top of Return stack
A	Address register
B	Address register

Remember that fetch pushes the stack, store and binary operations pop it.

Code	Op	Action
0	;	Return
1		
2	word	Call subroutine
3	word ;	Jump to subroutine (tail recursion)
4		
5		
6	if	Jump to 'then' if T0-T17 are zero
7	-if	Jump to 'then' if T17 is zero
8	n	Fetch literal
9	@+	Fetch from address in A; increment A
a	@b	Fetch from address in B
b	@	Fetch from address in A
c	!r	Store into address in R; increment R
d	!+	Store into address in A; increment A
e	!b	Store into address in B
f	!	Store into address in A

Code	Op	Action
10	+*	Add S to T if T0=1; shift right
11	2*	Shift T left 1 bit
12	2/	Shift T right 1 bit; preserve T17
13	-	Ones-complement T
14	+	Add S to T
15	or	Exclusive-or S to T
16	and	And S to T
17	drop	Move S to T
18	dup	Duplicate T
19	over	Fetch S
1a	pop	Fetch R
1b	a	Fetch A
1c	.	Do nothing
1d	b!	Store into B
1e	push	Store into R
1f	a!	Store into A

colorForth

A complete description is available at www.colorforth.com. The feature relevant here is that the color of a word determines its function:

- red - define new word
- green - compile
- yellow - execute
- white - comment

To avoid color for this paper:

- caps - define
- normal - compile
- bold - execute
- italic - comment

Bold will also be used when words are referenced in text.

Compiler

The c18 has 192 words of 18-bit memory. RAM addresses are 0-7f; ROM are 80-bf. Instructions are 5 bits long. There are 4 slots for instructions in each word. From left to right: S0, S1 and S2 are 5 bits; S3 is 3 bits and has 2 low-order zeros appended. The most common instructions end in 00.

The Pentium compiler interprets colorForth source and packs instructions into the low-order 18 bits of its 32-bit memory. There are several constraints:

- Only ...00 instructions fit in slot 3
- Jump instructions restricted to slots 0 and 1. Jump address is in slots 2 and 3
- Jumps are to slot 0 of destination

c18 source is green like Pentium colorForth. But green words are redefined to be executed. Each word compiles its instruction code into the next available slot. And green numbers are redefined to compile a c18 literal. Yellow words or numbers are not appropriate in c18 source, since c18 words cannot be executed.

In order to define c18 words that compile a call to themselves, the colorForth construct class is used. The code indicated by variable 'class ' is executed whenever a word is defined. For c18, this is a macro that compiles Pentium code that will push the c18 address onto the stack and call a word that compiles a c18 call.

Of course, 'empty ' is redefined to restore the normal behavior of green words, green numbers and class.

Note that jump instructions often force a new instruction word. Sometimes rearranging the code can fill otherwise empty slots. It's best to keep definitions small and simple and to be aware of how well instructions are packing.

Compiler code

Here is some selected code from the compiler. For those who can run colorForth, the 3 blocks of pre-parsed source is available at www.colorforth.com

```
S4 n   h @ ip ! 8192 * , 1 slot ! ;
S0 n   8192 *
SN n   dup call? ! ip @ +! 1 slot +! ;
S1 n   256 * sn ;
S2 n   8 * sn ;
S3 n   dup 3 and drop if 7 sn s4 ; then 4 / sn ;
I, n   slot @ jump s0 s1 s2 s3 s4
```

I, jumps to the code for the current slot. **Slot**, **h**, **ip** and **call?** are variables. **Ip** is the current instruction address, **H** is the next available address. **H** will be **ip**+1 unless a literal has been compiled. Slot 4 marks the end of the current instruction word. Most instructions are defined using **i,**. For example:

```
@    b i, ;
2*  11 i, ;
+   14 i, ;
.   1c i, ;
```

Literals require special attention. **N** is defined:

```
N   defer   8 f@ execute  8 i, 3ffff and , ;
```

Defer returns the address of the folowing code which will be assigned to short green numbers (all 18-bit numbers are short on a 32-bit host). Thereafter these will encounter the phrase **8 f@ execute** which extracts the number from its pre-parsed word as for a yellow short number. This returns the literal on the stack. Now the instruction code 8 is compiled and the literal clipped and compiled into the next word. Up to 4 literals can be referenced in the same instruction word, which is followed by the actual numbers.

Jump instructions use **adr** which compiles the instruction code and returns the address of the instruction. **Break** forces the following instructions into a new instruction word:

```
BREAK   4 slot ! ;
ADR n-a   slot @ -2 and drop if . adr ; then
   s3? if . adr ; then  i, ip @ break ;
CALL    defer a   ff and 2 adr +! ;
IF -a   6 adr ;
-IF -a   7 adr ;
THEN a   h @ ff and swap +! 0 call? ! break ;
UNTIL a   ff and 7 adr +! ;
```

Adr compiles **.** until slot 0 or 1 is available. **Until** jumps back until the stack becomes negative.

```
;  call? @ dup 4000 or drop if dup 200 or drop if
   0 and i, ; then then 2/ ip @ +! ;
```

; checks if the last instruction was a call. If so, it changes the call to a jump. This "tail recursion" saves a return instruction without requiring syntax to generate a jump. **call?** is a variable storing the last instruction compiled. It has been shifted, so implies the slot it occupies.

Target code

Here is an example of target code. This c18 is asked to read Flash and place it into internal RAM.

```
90 org
BIT --1  -1 64 begin  over   + until drop ;
@F  --1n    36 begin  bit b! + until b@ ;
WORDS na    a! begin  @f !+  + until drop ;
80 org  102 b!  63 0 words
```

This code takes advantage of the fact that **until** ends with -1 on the stack. **Bit** counts half a clock period. **@f** generates 18 clocks and reads the input shift register. **B!** in this case toggles the clock line. **Words** reads and stores the input words.

Some code will be standard in ROM. For example, shift and multiply routines:

```
*7 nn-nn   *+ *+ *+ *+
      *+ *+ *+ ;
2*15 n-n   2* 2* 2*
2*12  2* 2* 2*
2*9   2* 2* 2*
2*6   2* 2* 2*
      2* 2* 2* ;
2/15 n-n   2/ 2/ 2/
2/12  2/ 2/ 2/
2/9   2/ 2/ 2/
2/6   2/ 2/ 2/
      2/ 2/ 2/ ;
```

Calling these predefined routines trades time for space. A call takes 3-4 slots and 3 cycles. An in-line 2*15 would save the cycles, but take some 19 slots, since 2* can't be in slot 3. Of course, to shift left 8 places:

```
2* 2* 2*6
2* 2*6 2*
2*6 2* 2*
```

whichever packs best.