Freedom of Assembly

Julian V. Noble

January 5, 2001

Abstract

Many programmers have found the development and testing of assembly language subroutines easier within a Forth environment than using more conventional tools. Subroutines created this way can then be transformed to a form compatible with commercial assemblers, and linked to calling programs in mainstream languages. This article illustrates how to do this with three example subroutines of increasing complexity, using a popular public domain Forth running under MS Windows. The third example shows how to create tools not available with conventional assemblers.

Introduction

Programing in assembly language is no task for the faint of heart. Machine language programs are hard to compose, hard to get right, hard to understand, hard to maintain, and hard to port to other computers. By contrast, high level programming languages provide standardized data structures and operations that accomodate the needs of most users. Much effort has been devoted to developing optimizing compilers that generate machine code whose speed approaches the best efforts of wizard hackers. Such facts of life have led to a common notion that machine language programming is obsolete [1].

But sometimes we encounter problems with no reasonable high level solution. Interfacing external devices; trying to perform an operation trivial in machine code (but time-consuming, circuitous, or even impossible in a high level language); or seeking ultimate speed lead us—however reluctantly—to exercise our constitutional right to assemble.

Modern programming environments let us link high level programs with machine code procedures that have been assembled separately (that is, outside the compilation process), thereby combining the ease of high level programming with the advantages of assembler. The usual procedure is

1. program, test and debug everything possible in high level code (clearly this does not apply to operations not provided by the language);
2. using a profiler or algorithmic analysis, determine which portions can be rewritten profitably in machine language;
3. assemble, link and test the hand-coded parts;
4. repeat *ad nauseam.*

Step 3 is generally so arduous as to relegate assembly language to the category of desperate measures.

If only there were a way to test fragmentary assembly language subroutines in isolation—to assemble and run them as separate programs. This would be analogous to the ability of certain interpreted languages (*e.g.*, Basic, Lisp or Forth) to execute program fragments. Testing isolated machine code fragments telescopes the assemble—compile—link—test—debug cycle into a single stage, eliminating Step 3 entirely. Once satisfied with our machine code fragments we could concatenate them, (re)assemble them, and link them to the main program, testing the result at most once or twice more. To be sure, some environments provide "debuggers"—mini-assemblers that operate in an interpretive mode. For example, `DEBUG.EXE`, available on DOS and Windows™ systems, can assemble and single-step through simple 16-bit machine code routines. It can even create standalone `*.com` executables. But I would not try to replace `MASM`™ or `TASM`™ with `DEBUG.EXE` or its 32-bit cousins.

Forth programmers know a shortcut that makes machine language programming nearly as simple as high level programming. As it happens, Forth is my favorite language, but not everyone likes it. And often enough, constraints imposed by management preclude using Forth in commercial applications. But for assembling and testing isolated machine code fragments, Forth is unrivaled. Thus you should consider it for rapid development and testing, even if you intend to link the end product to Fortran, C or C++. And if you just want to learn assembly language, a Forth environment provides the most nearly painless way I have yet discovered.

Assembly coding in Forth is easy because in Forth everything is either a literal number or a subroutine (called a "word" in Forth-speak) that executes when it is named [2]. We define new subroutines either by threading together previously defined words ("high-level" definition) using the machine stack as a communications interface; or else by using the built-in assembler to define subroutines directly in machine code.

A Forth subroutine executes when it is typed in at the console. Subroutines defined in high-level Forth or in assembler appear identical to the user. Here is an example:

```
3 4 5 * + . <cr>   23 ok
```

The Forth interpreter seeks the strings 3, 4 and 5 in the dictionary of previously defined words, but fails (they are numbers, not subroutines). Therefore it converts them to numbers and pushes them on the stack. The words `*` and `+` are in fact subroutines (machine-coded primitives). The interpreter finds them in the dictionary and executes them successively, leaving the result (23) on the stack. The subroutine dot (`.`) consumes the top number on the stack and displays it to the console.

Alternatively I could define a new subroutine that concatenates multiplication and addition:

```
: *+    *   +   ; <cr> ok
```

If I then type in

```
3 4 5 *+ . <cr>
```

I will see the same result (23 and an `ok`) displayed to the console.

But suppose I preferred to define `*+` directly in machine language. I would type in (perhaps omitting the comments)

```
CODE *+             ( a b c -- b*c+a) \ stack: before -- after
    mov ecx, edx    \ save edx register because mul alters it
    pop eax         \ get item b; item c (TOS) is already in ebx
    mul ebx         \ integer multiply-- c*b -> eax (accumulator)
    pop ebx         \ get item a
    add ebx, eax    \ add c*b to a -- result in ebx (TOS) --done
    mov edx, ecx    \ restore edx
    next,           \ terminating code for Forth interpreter
END-CODE  ok

3 4 5 *+ . 23  ok
```

The above machine-language version (defined using `CODE` and `END-CODE`) appears the same to the user as the high level version defined with colon and semicolon. (Parts of the code are specific to the internal workings of the Forth I am using. I will discuss this further below.)

Some C/C++ compilers provide in-line assemblers: for example, Microsoft C/C++ 7.0+ permits machine code in-lining using the compiler directive `__asm{`, whereas Borland compilers provide the plain directive `asm` [3]. Forth does much more than mere code in-lining. First, Forth provides all the power of a stand alone assembler (which the C/C++ in-line assembler certainly does not). If a feature you are used to is missing from the Forth assembler, you can always add it because Forth is extensible. Second, assembly in Forth is direct. When you type in a `CODE` subroutine, it compiles instantly and is then ready to run from the command line. With C or C++ you must compose, compile and execute a more elaborate complete program, that includes both your in-line code and a program to exercise it, before you can tell whether it works.

Forth assemblers are written in Forth, hence operate the same way as any other set of Forth words. Assembler mnemonics such as `mul ebx` install the op-code(s) of the corresponding machine instructions in the body of a new `CODE` definition. The examples in this article were composed using the popular Windows-based, public domain Win32Forth [4], whose built-in assembler is Jim Schneider's `486asm.f`; however, assembly code works similarly in all Forths, with only minor differences arising from implementation details.

Win32Forth's decompiler exhibits the machine code of its primitive kernel words, a major pedagogical advantage. Primitives provide convenient examples both of the assembler's operation, and of how to use Intel 80486+ machine code to perform simple operations. Working within the Win32Forth environment provides greater than average safety, since memory access is in protected mode. For example, my first attempt at defining `*+` in `CODE` was

```
CODE *+             ( a b c -- b*c+a) \ stack: before -- after
    pop eax         \ get item b; item c (TOS) is already in ebx
    mul ebx         \ integer multiply-- c*b -> eax (accumulator)
    pop ebx         \ get item a
    add ebx, eax    \ add c*b to a -- result in ebx (TOS) --done
    next,           \ terminating code for Forth interpreter
END-CODE  ok
```

When I tried to run this version it failed, with the protected-mode exception message:

```
.....................................................................

EXCEPTION: ACCESS_VIOLATION
  Registers:
    Eax:          5538h
    Ebx:             Ch   top of data stack
    Edx:             0h
    Ecx:        77553Ch
    Esi: IP       555Ch   image relative
    Edi: IMAGE  770000h   Forth's base address
    Esp: SP@  FFEDDDD0h   image relative
```

```
    Ebp: RP@  FFEDFCF4h   image relative
    Eip: PC       553Dh   image relative
RETURN STACK[4]: ?STACK+0 _INTERPRET+21 QUERY-INTERPRET+2 CATCH+14

  At forth word:    DEPTH
```

.....

To find the bug, I decompiled the Forth primitives `*` and `+` by typing in "`see *`" and "`see +`", to which the system responded

```
* IS CODE
          1490 8BCA             mov     ecx , edx
          1492 58               pop     eax
          1493 F7E3             mul     ebx
          1495 8BD8             mov     ebx , eax
          1497 8BD1             mov     edx , ecx

         ok
```

and

```
+ IS CODE
           EC0 58               pop     eax
           EC1 03D8             add     ebx , eax

         ok
```

It turned out that I had forgotten that integer multiplication spills over into the `edx` register. But Win32Forth wants this register (and several others) to be preserved (otherwise it returns control to some random place in memory). Obviously one must save `edx` before doing anything else, and restore it just before leaving the subroutine.

There is a moral here: assembly language programming is like walking a high wire without a safety net. It is absolutely essential to know what infor-

mation a calling program expects to have preserved by a called subroutine; as well as how any instruction affects the state of the cpu.

This article provides three examples of machine code programming in a Forth environment: the subroutine `UCASE` converts all lower-case letters in a string to upper case, leaving digits and punctuation alone; `}}inmostLU` replaces the inner loop of the LU algorithm; and `new_point` calculates a step in the Runge-Kutta solution of a ballistics problem. The third example is especially interesting because it led me to add macro capabilities as well as a simple formula parser to the existing assembler.

In what follows I assume the reader is familiar with Intel 80x86 assembly language mnemonics. These can be found in various Intel publications [5], as well as in books on assembly language programming for the Intel family of chips [6].

Case conversion

Many libraries provide a function for converting a string to all upper case, or all lower case letters, leaving digits and punctuation alone. The new Forth ANS standard [7] happens not to require such a routine, although most Forths employ a word analogous to `UCASE` in their compiling mechanisms.

The first step is to choose our approach. Different languages manage strings in different ways. Such linguistic differences demand custom-tailored `CODE` subroutines. Additionally, machine code subroutines intended for linking to programs in high-level languages require language-specific "boiler-plate" headers and footers. The language reference manual usually specifies such boiler-plate.

For simplicity we use the C convention: we store an ASCII string of N characters as $N+1$ contiguous bytes in memory, with the $N+1$ 'st (terminal) byte set to 0. We reference a string by the address of its first character. To decide how to proceed, we write the case-changing subroutine initially in high level Forth. (The auxiliary Forth words `$0"` and `$0.`—respectively, to create and display 0-terminated strings—are non Standard so I define them

in Appendix A.) Many languages implement string-translation functions like UCASE with a translation table. The routine simply steps through a string character by character, replacing each by the corresponding value from the table. I exhibit this algorithm below:

```
: char_table:    ( size --) \ create an ASCII character table
     CREATE   0 DO  I C,  LOOP  ;
: replace        ( char.n' char.0' char.n char.0 base_adr -- )
\ replace a seqence char.0-char.n in a table by char.0'-char.n'
     LOCALS| base_adr char.0 char.n char.0' offset |
     char.0'  char.0  -  TO  offset
     char.n 1+  char.0                      \ loop limits
     DO   I  offset +   I base_adr +  C!  LOOP  ;

 256 char_table: ucase_table
 char Z  char A   char z  char a  ucase_table  replace

 : >ucase   ( offset -- char)  CHARS  ucase_table  +  C@  ;

 : ucase      ( c-adr -- )
    LOCALS| adr |       ( -- )
    BEGIN               \ work thru string from left
        adr C@          ( -- char)    \ get character
        ?DUP            ( -- char char | 0 ) \ dup if <> 0
    WHILE               ( -- char)  \ skip if TOS = 0
        >ucase          ( -- char')
        adr   C!        \ replace modified character
        adr 1+  TO adr  ( -- )    \ incr. adr
    REPEAT              \ loop back to BEGIN
 ;
 \ see Appendix A for $0" and $0.
 $0" this + is A % lcase STring"  UCASE  ok
 pad $0. THIS + IS A % LCASE STRING ok
```

The table method translates easily to assembler, for example using the special Intel instruction `xlat`, but we pursue it no further because standard references cover it adequately.

If we cannot spare 256 bytes of memory for a translation table, we can test whether each character is a terminating 0, a lower case letter or "other". If it is 0, exit—we are done. If it is a lower case letter, change it to upper case; otherwise do nothing. The actual switch from lower– to upper case can be accomplished by subtracting 32d from the ASCII character code of the letter, since the upper case letters have codes 32d smaller than their corresponding lower case values:

```
\ environmental dependence: assumes ASCII encoding

: lcase?           ( char -- flag)  \ true if lower case
   DUP  [CHAR] a < ( char f1)       \ true if char < "a"
   SWAP [CHAR] z > ( f1 f2)         \ true if char > "z"
   OR              ( not[flag] )    \ combine flags
   NOT ;                            \ logical not

: ucase      ( c-adr -- )
    LOCALS| adr |     ( --)
    BEGIN             \ work from left to right thru string
        adr C@        ( char)          \ get character
        DUP 0<>       ( char flag)
    WHILE
        DUP lcase?    ( -- char flag)
        32 AND        ( -- char  n=[32 AND flag] )
        -             \ subtract 32 from lcase letters
        adr C!        \ replace modified character
        adr 1+ TO adr ( --)  \ inc adr
    REPEAT            \ loop back to BEGIN
;
```

Note that the words `lcase?` and `ucase` *compute* results rather than deciding them [8]. That is, `ucase` could have been coded with a branch:

```
lcase?    IF    32 -   adr  C!    ELSE   DROP   ENDIF
```

Avoiding the branch means we replace every character, not just those whose case we change. (The translation-table approach shares this disadvantage.)

However, tests show that text input is predominantly lower case; the time consumed in the hypothetical branch dominates the superfluous store operations on many computers. (On older cpu's branches consume many machine cycles because they stall the pipeline. Avoiding branches then really saves time. But advanced cpu's have multiple pipelines and branch prediction, so branch avoidance may, paradoxically, be slower than an explicit branch—this is why you must test every idea.)

Now test our Forth subroutine:

```
$0" this + is A % lcase STring"  UCASE
PAD $0. THIS + IS A % LCASE STRING ok
```

We translate to assembly language in parts, beginning with `lcase?`. Note that in Win32Forth, as in many other Intel-based Forths, the top of the stack (TOS) is cached in the `ebx` register; the first `pop ebx` instruction is therefore superfluous in Forth, but will be necessary in a C-compatible subroutine. We give the `CODE` routine a different name so we can compare it with the high-level one:

```
CODE lcase?C ( char -- flag) \  timings:         Pentium  486
\   pop ebx                  \ necessary with C    1       1
  mov eax, ebx               \ copy TOS to accum.  1       1
  sub eax, # char a 1-       \ char  < a           0       1
  mov ecx, edx               \ save edx            1       1
  sub ebx, # char z 1+       \ char  > z           0       1
  xor  eax, ebx              \ neg iff either neg  1       1
  cdq                        \ sign eax -> edx     2       3
  mov ebx, edx               \ edx to TOS          1       1
  mov edx, ecx               \  restore edx        1       1
                             \   total cycles:     8      11
  next,
END-CODE
```

Test it (note that "true" has all bits set and therefore is displayed as –1 in systems with 2's-complement arithmetic):

```
CHAR A DUP . lcase?C . 65 0 ok
CHAR a DUP . lcase?C . 97 -1 ok
CHAR z DUP . lcase?C . 122 -1 ok
CHAR & DUP . lcase?C . 38 0 ok
```

The subroutine works as advertised.

We would next like to be sure the `CODE` version is indeed faster than the high-level one. Thus we employ the software timer in Win32Forth to time multiple repetitions (a software timer running under Windows cannot achieve greater precision than about ± 0.05 seconds; on a 150 MHz machine this uncertainty represents $\pm 4 \times 10^4$ iterations of the slowest version of this code):

```
\ Tests performed on a 150 MHz Pentium-class machine.
CODE nuttin'     \ do-nothing code to time an empty loop
    next,
END-CODE

: test0   time-reset          \ empty loop
          0 do  [char] a  nuttin'  drop  loop  .elapsed  ;
: test1   time-reset          \ hi-level test
          0 do  [char] a  lcase?   drop  loop  .elapsed  ;
: test2   time-reset          \ code version
          0 do  [char] a  lcase?C  drop  loop  .elapsed  ;

100000000 test0 Elapsed time: 00:00:22.740 ok
100000000 test1 Elapsed time: 00:02:03.530 ok
100000000 test2 Elapsed time: 00:00:26.750 ok
```

We see that 10^8 repetitions of the high-level code takes 101 seconds (exclusive of loop bookkeeping, putting a number on the stack, and dropping it again), whereas the `CODE` version runs in 4.0 sec, about 25 times faster. The number of cpu cycles per iteration of `lcase?C` is about 6, suggesting the parallelism of the cpu is slightly better than advertised.

To continue translating `UCASE` we must decide how to implement a loop. We can synthesize an indefinite loop using tests and jumps (`jz`, `jnz`, *etc.*);

or a definite loop using the Intel `loop` instruction. The latter would be appropriate for Forth- or Basic-like counted strings, but not for C-like 0-terminated strings. Therefore we employ jumps:

.....

```
HEX                             \ switch base to hexadecimal
CODE ucaseC                     ( 42 bytes)
        \ header section
        \ pop ebx               \ needed in C version
        push edx                \ save edx -- maybe unneeded in C

        \ body section                    timings:  Pentium  486
L$1:    mov cl, [ebx] [edi]     \ get character       1       1
        and ecx, # 000000FF     \ set bits 8-31 to 0  1       1

             \ test section
        jz  L$2                 \ if char=0, exit     1       3
        mov eax, ecx            \ char -> accum.      1       1
        sub eax, # char a 1-    \ char  < a ?         1       1
        sub ecx, # char z 1+    \ char  > z ?         1       1
        xor  eax, ecx           \ neg if either neg   1       1
        cdq                     \ sign eax -> edx     2       3
             \ end test section     edx = flag

             \ replace character section
        and edx, # 20           \ 32 if lcase, 0 else 1       1
        sub byte [ebx] [edi], dl                  \   2       3
             \ end replace char section

        inc ebx                 \ ebx = ebx + 1       1       1
        jmp L$1                 \ unconditional jump  1       3
        \ end body section         back to L$1

        \ footer section
L$2:    pop edx             \ restore edx may not be necessary
        pop ebx             \ pop to TOS -- delete in C version
        next,               \ absent from C version
```

```
END-CODE
DECIMAL                          \ switch base back to decimal
```

. . . . . .

The body of the subroutine is hard to follow even with detailed comments (which is why we prefer high level language to assembler), but it is essentially the same as the high level definition: step through the string, testing and replacing lower-case letters, until it encounters the terminal 0. A machine code subroutine additionally needs a preamble, or "header", that saves registers that need saving, and places initial data in appropriate registers. It also needs an epilogue, or "footer" that restores registers and exits gracefully.

Testing and timing as before, we have

```
\ Tests performed on a 150 MHz Pentium-class machine.

$0" this + is A % lcase STring"  ucasep  ok
pad $0. THIS + IS A % LCASE STRING ok

: test0    time-reset   0 DO  DUP  DROP  LOOP  DROP  .elapsed ;  ok
: test1    time-reset   0 DO  DUP  ucase  LOOP DROP  .elapsed ;  ok
: test2    time-reset   0 DO  DUP  ucasec LOOP DROP  .elapsed ;  ok
pad 10000000 test0 Elapsed time: 00:00:01.750 ok
pad  1000000 test1 Elapsed time: 00:00:51.360 ok
pad 10000000 test2 Elapsed time: 00:00:21.470 ok
```

The tests give running times of about 5.1×10^{-5} and 2.0×10^{-6} seconds per iteration, respectively, for the high-level and assembler versions, roughly a factor of 26 speedup. The time per character in the assembler version is about 7.6×10^{-8} seconds, or about 11 machine cycles. The clock count is 14, so again the cpu is slightly better than advertised. The translation table method (using `jump`s rather than `xlat`s) executes in 6-8 cycles per character but requires about 240 more bytes of storage—yet another example of the tradeoff between storage space and execution time.

Inner loop in LU

The LU algorithm for solving linear equations [9] has an inner loop that appears in two places. The high level Forth code for this loop is

```
: }}inmostLU    ( a{{ I0 J0 Limit --)     ( f: sum -- sum')
    LOCALS| Limit J0 I0 a{{ |
    Limit  0  DO   f" a{{ I0_I }} * a{{ I_J0 }}"  F-    LOOP  ;
```

The above code, simple though it is, becomes rather involved when we try to translate it to Intel machine code because of the paucity of available registers. It therefore behooves us to compute certain addressing constants *before* entering the loop; that is, we should rewrite }}`inmostLU` to expect these constants on the stack.

```
: }}inmostLU   ( f: sum -- sum')
    ( a[I0,Limit] a[Limit,J0] #cols*length length Limit --)
    \ length is the size of the matrix element in bytes
    \ f" sum = sum - a{{ I0_I }} * a{{ I_J0 }}"
    LOCALS| Limit step1 step2 base_adr1 base_adr2 |
    0 Limit  DO    base_adr2 F@   base_adr1 F@   F*   F-
                   base_adr2 step2 -  TO base_adr2   \ dec base_adr2
                   base_adr1 step1 -  TO base_adr1   \ dec base_adr1
    -1 +LOOP
;
```

When writing the assembler version we can then put as many as possible into registers, accessing the rest directly from the stack. Another point is that }}`inmostLU` embodies a definite loop so we can use the Intel `loop` instruction. The count is placed in the `ecx` register and is decremented on each iteration until it reaches zero. So we should rewrite the high level version to reflect this also. Finally, we need to keep in mind the array storage convention. The Forth Scientific Library project has adopted the C-like convention that arrays are stored row-wise, rather than column-wise as in Fortran. Thus if we wish to link this subroutine with a Fortran version of the LU algorithm,

we would need either to transpose the matrix before beginning; or better, to rewrite }}inmostLU to assume column-wise arrays.

The assembler version will then have the structure

```
CODE  }}inmostLU    ( f: sum -- sum')
        ( a[I0,Limit] a[Limit,J0] #cols*length length Limit --)
        \ this version uses 64-bit reals

        \ header section
        \ ....

        \ loop section                    timings:        Pentium   486
L$1:    fld  [eax] [edi]      ( fpu: sum a[I0,I])          1        3
        fmul [ebx] [edi]      ( fpu: sum a[I0,I]*a[I,J0]) 3/1      14
        sub eax, esi          \ eax = edx - #cols*length   1        1
        sub ebx, edx          \ ebx = ebx - length         0       1/3
         \ subtract because we are counting down!
        fsubp st(1), st       ( fpu: sum')                3/1      8-20
        loop L$1                                          5/6      6/7

        \ footer section
        \ ....
END-CODE
```

The actual operations contained within the loop are simple: get one matrix element, multiply it by another, and accumulate the products on the fpu stack. The loop comprises a fetch, a multiply and an addition. The complete code for this subroutine appears in Appendix B.

The innermost loop in the LU algorithm contains the only instructions executed $\mathcal{O}(N^3)$ times. Everything else is executed $\mathcal{O}(N^2)$ times or fewer. By rewriting this loop in machine code, we can solve large dense systems as fast as or faster than, the code produced by the best optimizing compilers. That high level Forth is 3 to 10 times slower than optimized Fortran or C then becomes irrelevant, since the $\mathcal{O}(N^3)$ term dominates the running time.

Ballistics problem

In my course, *Computational Methods of Physics*[1], I have been using a ballistics problem as a case study in the numerical solution of differential equations, as well as function minimization. A cannon located at a specified latitude, longitude and altitude fires a round shot of specified mass, diameter and muzzle speed. The projectile is realistic: it experiences both air resistance (that varies with air density, hence with altitude) and the Coriolis effect of the Earth's rotation. The problem is to find the elevation angle θ and direction of aim, φ that will hit a target whose distance, height and direction are known, within a given tolerance[2].

In order to achieve acceptable execution speed for experimenting with various algorithms, I resorted to assembly code. Although the problem is not in itself of general interest, I discuss it here because it led me to create tools that might be helpful to anyone facing a big assembly job.

To simplify the problem, ignore the Coriolis effect (so that only the muzzle elevation angle θ need vary); the equations of motion are then

$$\begin{aligned} \dot{v}_x &= -D(y)vv_x \\ \dot{v}_y &= -D(y)vv_y - g \\ \dot{x} &= v_x \\ \dot{y} &= v_y \,, \end{aligned}$$

where $D(y)$ is the drag function and $\dot{x} = \frac{dx}{dt}$. To reduce by one the number of equations, change the independent variable from t to x:

$$\frac{d}{dt} \to \frac{dx}{dt}\frac{d}{dx} \equiv v_x \frac{d}{dx} \,,$$

leading to

$$\frac{dv_x}{dx} = -D(y)v$$

[1] the URL for this course is `http://Landau1.phys.virginia.edu/classes/551/`

[2] This is where the minimizer comes in!

$$\begin{aligned} \frac{dv_y}{dx} &= -D(y)v\frac{v_y}{v_x} - g \\ \frac{dy}{dx} &= \frac{v_y}{v_x}\,. \end{aligned}$$

The inner routine, that advances the differential equation solver by one step, was all that had to be translated to assembler. Here is what it looks like in high-level Forth (using a FORmula TRANslator for clarity):

```
\ FORmula TRANslator: text between f" and " is treated
\ as a formula, parsed and translated into Forth, and compiled.
\ Thus f" kx = -xstep * drag1 " compiles the Forth code
\         xstep F@ drag1 F@ F* FNEGATE kx F!

: new_point
  \ drag function
    f" drag1 = D(y) * sqrt(Vx^2 + Vy^2)"

  \ 1st Runge-Kutta step
    f" kx = -xstep * drag1 "
    f" ky = -xstep * (drag1 * Vy + g)/Vx"
    f" dy = xstep*Vy/Vx"

  \ drag function'
    f" drag2 = D(y+dy) * sqrt( (Vx+kx)^2+(Vy+ky)^2 )"

  \ 2nd Runge-Kutta step for V's
    f" (ky - xstep * (drag2 * (Vy + ky) + g)/(Vx + kx))/2"
    f" (kx - xstep * drag2)/2"

  \ update velocities
    Vx f+!
    Vy f+!

  \ 2nd Runge-Kutta step for y
    f" (dy + xstep * Vy/Vx)/2"
```

```
  \ update coords
    y f+!

  \ optionally compute flight time: f" t = t + xstep/Vx"
    f" x = x + xstep"
;
```

Clearly, converting so many equations to assembler is quite a chore. This led me to invest in a time- and energy saving mini-compiler. As I had already written the FORmula TRANslator employed above, it seemed a small additional step to write a translator that would generate Intel assembler mnemonics, suitable for incorporating into `CODE` definitions. It turned out to be surprisingly easy to do this. (The only difficulty I encountered was a mental blind spot: I persistently overlooked the fact that `dx` is the name of a register, so it cannot be used as the step size in an assembly language differential equation solver!)

I call the code translator `CTRAN`. It can be found on the *Computational Methods* Web page[1], under the heading *Forth system and example programs*. Basically, `CTRAN` provides two facilities to simplify creating large blocks of assembly code:

- assembler macros to assemble functions inline;
- translation of simple algebraic formulas into assembler mnemonics.

To create the `CODE` version of `new_point` I replaced all the `f"`s with `cf"` ; and also replaced the initial `:` by `CODE` and the terminating `;` by the phrase `next,~END-CODE`. In the interest of simplicity `CTRAN` does not translate functions or numeric literals. That is, `CTRAN` translates simple arithmetic expressions incorporating ($+$, $-$, $\times$, $\div$) and of course, arbitrary levels of parentheses. In the code that follows, `drag` and `(f!)` are macros. (The macro facility is described in Appendix D, including definitions of `drag` and `(f!)`.)

Putting all the pieces together, the `CODE` version of `new_point` becomes

```
CODE new_point
    \ Note: it is much faster to multiply than to divide on
    \ the Pentium fpu.

    cf" Vx"  cf" Vy"  cf" y"     \ 1st Runge-Kutta step
    drag  (f!) drag1                \ drag1 = drag(Vx,Vy,y)

    cf" kx = -xstep * drag1"
    (1/f) Vx  (f!) invVx            \ invVx = 1/Vx
    cf" dy = xstep * Vy * invVx"    \ -- eliminates 1 division
    cf" ky = -xstep * ((drag1 * Vy) + g) * invVx"

    cf" Vxp = Vx + kx"           \ 2nd R-K step
    cf" Vyp = Vy + ky"
    cf" Vxp"  cf" Vyp"  cf" y+dy"
    drag  (f!) drag2                 \ drag2 = drag(Vxp,Vyp,y+dy)

    cf" Vx = ( kx - xstep * drag2 ) * half + Vx"
    cf" Vy = (ky - xstep * ((drag2 * Vyp) + g)/Vxp) * half + Vy"
    cf" y  = ( dy + xstep * Vyp/Vxp ) * half + y"

    cf" x = x + xstep"               \ update coords
    \ cf" t = t + xstep/Vx"          ( optional)

    next,
END-CODE
```

I assure the reader the above has been tested, really works, and reduces the run time about 11-fold. Decompiling *via* `see new_point` produces two pages of vertical-format assembly code. It would have been quite a chore to define and test this much code, but translating formulas with `cf"` reduced the subroutine to manageable proportions. The result is also far more readable than assembly code, no matter how well formatted and commented.

Your right to assemble

To summarize, assembly-coded routines have two advantages over high-level routines: they can be faster; and they can enable you to do things the designer forgot to include in your favorite language. Assembler has several disadvantages as well. Foremost is lack of portability: machine code routines must be rewritten for each new operating system and each new cpu. Second, they are harder to maintain than high level code. And finally, they are harder to compose and debug.

Using Forth to develop and test assembly language routines interactively takes much of the sting out of the issues of design, test and maintainance. The }}`inmostLU` example exhibited the value of rewriting the high level subroutine to simplify the data structures, thereby simplifying the algorithm. (I suspect few if any optimizing compilers can redesign your algorithm to improve the efficiency of its machine code.) The ballistics problem showed how macros extend the capabilities of the assembler. But I have never heard of a standalone assembler that incorporates an expression parser, or whose macro facility lets you define one. The parser and macro facility condensed the subroutine `new_point` from two pages of raw assembly code to a half page of formulas that can be understood at a glance. That is, good tools simplify maintainance and documentation.

Since assemblers written in Forth are rather simple, it is easy to modify them to recognize the mnemonics for—and generate the op-codes of—another cpu. That is, Forth systems are often used for *cross assembly*, with the resulting code being downloaded *via* a serial link to the "target" cpu. In fact, many Forths have been "meta-compiled" (that is, ported from one host to another) by first defining a simple cross assembler, then defining code primitives for the target machine. Once the kernel of the new Forth runs on the new machine, it is a simple matter to recompile the higher level words that define a fully functional system.

Suppose you are ready to code something following the prescriptions of this article. If your aim is speed I should caution you that good optimizing compilers produce code no worse than 1.5 to 2 times slower than the tightest assembly code. (Greater improvements can be obtained with code

intended to run on Intel chips and their clones, than with code for RISC chips. Intel CISC chips have few registers and arcane instruction sets, thereby offering greater scope for cleverness. I discuss some optimization issues for Pentium™-class chips in Appendix D.) The old adage that it is usually better to seek an improved algorithm than to optimize an existing one remains true. (Of course if you are interfacing with a device or adding a function to your programming language, this warning does not apply.)

Most compilers can output assembly listings generated during compilation. Simply include an appropriate compiler "switch" (such as `TASM`'s –S) when you invoke the compiler from the command line. Studying an assembler listing is a good idea for two reasons: first, to learn about your system's requirements, register preservation, *etc.*; and second, if your goal is optimizing for speed, to see whether there is any point—if your compiler already produces better code than you can, why bother?

Unless you are actually working in Forth, or using a Forth system for cross-assembly, your interest in the Forth environment is primarily for developing and testing the concept. Eventually you must translate the Forth-compatible file into something you can feed a stand-alone assembler to produce object code linkable to your preferred language. What I cannot tell you (but you need to find out) is how a high level program in this language calls a subroutine. That is, where does it put its arguments, which machine registers must it preserve, where does the result go? Will you be working in a true 32-bit environment or using 16-bit instructions? Are there long jumps (more than 128 bytes backward or 127 bytes forward) in your subroutine? This knowledge allows you to write the boiler-plate sections—the standard header and footer—that instruct the assembler to create a subroutine appropriate to your needs [1, 3].

Any text editor can replace the Forth comment delimiter \ with the usual assembly-code comment delimiter `;` . I have emphasized that one should write routines in three sections: header, body and footer, where the header and footer are language-specific and meant to be replaced. But there may be other things you will have to modify to conform with your cpu and operating system. For example, even if your system is a Pentium running under Windows, named variables in your system may be stored somewhere other than the memory segment that Win32Forth references by the offset `[edi]`.

(In Intel-ese `[edi]` tells the assembler to add the contents of the register `edi` to an address.) A Fortran- or C-linkable subroutine's arguments (as well as local variables) will be stored in a "stack frame" and accessed by stack offsets. It pays to study examples appropriate to your own circumstances, for example from the assembler output of a compiled program.

All the high level code in this article will run without modification on any Forth compliant with the 1994 ANS Standard. The macro generator included with `CTRAN` is generic. The parser itself, however, outputs Intel code, optimized for Pentium-class cpu's, and conforming to the calling conventions of Win32Forth. Hence you will have to modify the code generating sections if you have a different cpu or Forth system in mind. (This is not hard—it is a short program and I have marked the relevant places.)

Before I discovered Forth I spent endless hours developing and testing assembly code subroutines in the conventional manner. All of these subroutines were very simple—I would never have dared to tackle anything as complicated as the ballistics problem. Using a Forth-based assembler, the most complex subroutine I have yet written, a fast routine for Bessel functions, took less than one hour from start to finish. The examples in this article took even less time (not counting the time to write and test `CTRAN`—about 2 hours). I hope my experience encourages you to try the Forth route to rapid code development.

Appendix A: C-style string functions

```
\ Forth subroutines for experimenting with C-style strings

: get_len   ( beg -- len)
    DUP                 ( beg beg)
    BEGIN           \ start indefinite loop
        DUP C@          \ get char
        0<>             ( beg adr flag)
    WHILE   1+          ( beg adr+1)
    REPEAT              ( beg end+1)
                    \ loop until character is 0
    -  NEGATE           ( -- len)   \ compute length
;

: $0"   ( -- adr)       \ create 0-terminated string
    [CHAR] "  WORD      \ get input
    DUP                 ( $adr $adr)
    1+  PAD  ROT  C@    \ get length
    DUP >R              \ save it temporarily
    CMOVE               \ move text to scratchpad
    PAD  R>  OVER +     ( -- beg end+1)
    0 SWAP  C!          \ terminate with 0
;

: $0.   ( adr --)       \ print 0-terminated string
    DUP  get_len  TYPE  ;
```

Appendix B: CODE definition of }}inmostLU

The register-starved Intel architecture imposes some juggling in the "preamble" section of the code. The `loop` instruction has been replaced by the 3× faster sequence `dec ecx  jnz L$1`.

```
CODE  }}inmostLU    ( f: sum -- sum')
        ( a[I0,Limit] a[Limit,J0] #cols*#b #b Limit --)
        \ this version uses 64-bit reals
        \ header section
        fld FSIZE FSTACK_MEMORY  \ f: -> fpu:
        pop ecx                  \ ecx = Limit
        pop eax                  \ eax = #b
        push esi                 ( base2 base1 step2 esi)
        push edx                 ( base2 base1 step2 esi edx)
        mov esi, 8  [esp]        \ esi = #b * #cols
        mov ebx, 12 [esp]        \ ebx = a[Limit,J0]
        mov edx, 16 [esp]        \ edx = a[I0,Limit]
        \ loop section                    timings:        Pentium   486
L$1:    fld  [edx] [edi]         ( f: sum a[I0,I])          1       3
        fmul [ebx] [edi]         ( f: sum a[I0,I]*a[I,J0])  3/1     14
        sub edx, esi             \ edx = edx - #b * #cols    2       2
        sub ebx, eax             \ ebx = ebx - #b            2       2
         \ subtract because we are counting down!
        fsubp st(1), st          ( f: sum')                 3/1     8-20
        dec ecx                                             1       1
        jnz L$1                                             1      1/3
        \ footer section
        pop edx                  \ restore saved registers
        pop esi
        add esp, # 8             \ clean off stacks
        pop ebx
        fstp FSIZE FSTACK_MEMORY \ fpu: -> f:
        next,
END-CODE
```

Appendix C: example macro definitions

```
\ code for defining macros

: eval"   POSTPONE  S"  POSTPONE EVALUATE  ; IMMEDIATE

: (f!)    \ store from fpu to a named fp variable
          eval" fstp  FSIZE"
          BL  TEXT  COUNT  EVALUATE
          eval" [edi]"  ;

: (f+!)   \ add and pop fpu to a named fp variable
          eval" fadd FSIZE "
          BL TEXT COUNT  2DUP   EVALUATE
          eval" [edi]  fstp  FSIZE "
          EVALUATE
          eval" [edi] "  ;

FVARIABLE  two    2e0   two  F!
FVARIABLE  half   0.5e0 half F!

: (f2*)     \ multiply fpu TOS by 2
            eval" fmul  FSIZE  two   [edi] "  ;
: (f2/)     \ divide fpu TOS by 2
            eval" fmul  FSIZE  half  [edi] "  ;


\ drag function  ( note: inv_yscale = 1/yscale)
: drag    ( 87: Vx Vy y -- drag~D1*V*exp[-y*inv_yscale])

    eval" fmul FSIZE inv_yscale [edi] " ( 87: Vx Vy u=y*inv_yscale )
    eval" fld1                        "
    eval" fchs                        " ( 87: Vx Vy u  -1)

    eval" fld     st(1)               " ( 87: Vx Vy u  -1 u)
    eval" fmul    FSIZE half [edi]    " ( 87: Vx Vy u  -1 u/2)
    eval" faddp   st(1), st           " ( 87: Vx Vy u -1+u/2 )
```

```
    eval" fmulp   st(1), st           "
    eval" fld1                        "
    eval" faddp   st(1), st           " ( 87: Vx Vy 1-u+u^2/2 )

    eval" fxch    st(2)               " ( 87: 1-u+u^2/2 Vx Vy )
    eval" fmul    st                  " ( 87: 1-u+u^2/2 Vx Vy^2 )
    eval" fxch    st(1)               "
    eval" fmul    st                  "
    eval" faddp   st(1), st           " ( 87: 1-u+u^2/2 Vx^2+Vy^2 )
    eval" fsqrt                       " ( 87: 1-u+u^2/2 V )
    eval" fmulp   st(1), st           "
    eval" fmul    FSIZE D1  [edi]     " ( 87: drag )
;
```

Appendix D: Remarks on code optimization

The Intel Pentium™ processor and its clones (AMD, Cyrix) has two instruction pipelines, labeled U and V, and employs branch-prediction algorithms to guess intelligently which branch a process will take. This enables it to avoid stalling most of the time—that is, having to flush all pre-fetched instructions from the queue and reload. Thus, in contrast with its predecessors, programmers concerned to optimize code for the Pentium class of cpu's need not avoid branches resulting from decisions and loops.

Having two pipelines permits some parallelism of execution: certain combinations of instructions can be "paired" to execute simultaneously. Schmit [3] describes a number of ways to achieve this pairing for optimal results. The rules for pairing are somewhat arcane, and it is well to have Schmit's book or a similar reference handy.

The floating point unit offers less scope for optimizations exploiting parallelism. Floating point arithmetic instructions can be paired only with the `fxch` instruction (and not all can be paired). The key points to remember are:

- On the Pentium, multiplying (or adding/subtracting) the fpu stack-

top, `st(0)`, by a memory operand is as fast as multiplying it by another stack element (such as `st(2)`, *e.g.*). Therefore many loading operations can be dispensed with by writing

```
fld  FSIZE a [ebi]
fmul FSIZE b [ebi]
fadd FSIZE c [ebi}
```

to compute $a \times b + c$, rather than

```
fld   FSIZE a [ebi]
fld   FSIZE b [ebi]
fmulp st(1), st(0)
fld   FSIZE c [ebi]
faddp st(1), st(0)
```

The first is $\frac{5}{3}$ faster.

The expression to `CODE` parser, `CTRAN` has been designed to incorporate this optimization.

- Floating point division is about 39 times slower than multiplication on a Pentium. This is like the old IBM 704 I learned Fortran on in 1960: I was cautioned never to divide by a constant within a loop, but to multiply by a variable holding the inverse of the constant, that had been evaluated outside the loop. Given the enormous disparity in execution time on the Pentium, this is still good advice. In fact, it probably was not a good idea to change variables from t to x in the ballistics problem, since the multiplications and additions involved in solving a fourth differential equation are not nearly as time consuming as the two divisions the "simpler" algorithm imposes!

- The quondam "fast" instruction `fscale` for multiplying or dividing by a power of 2 should be avoided. It requires loading an integer constant (the power) which will usually be popped from the fpu stack afterward; executing `fscale` itself takes 21-32 clocks. Multiplying by the appropriate power of 2 stored in memory is therefore much faster (1 clock). The macros in Appendix C were designed with this in mind.

- A final optimization I have found useful is using the `lea` ("load effective address") instruction both for address arithmetic and for multiplying integers by certain integers. The latter use is described in Schmit's book.

References

[1] See, *e.g.*, Michael Abrash, *The Zen of Code Optimization* (The Coriolis Group, Inc., Scottsdale, AZ, 1994) for an eloquent defense of assembly language *vs.* high level language.

[2] J.V. Noble, "Adventures in the Forth Dimension", *Computing in Science and Engineering*, **2** #5 (2000) 6.

[3] Michael L. Schmit, *Pentium Processor Optimization Tools*, Academic Press, Inc. (Cambridge, MA, 1995).

[4] Win32Forth is the brain-child of Tom Zimmer with contributions from Andrew McKewan, Jim Schneider, Robert Smith Y.T. Lin and Andy Korsak. It is readily available from the various Web sites, including *http://www.taygeta.com*

[5] i486™ *Microprocessor Programmer's Reference Manual* ©Intel Corporation 1990 (Osborne/McGraw-Hill, New York, 1990).

[6] See, *e.g.*, L. Scanlon, *Assembly Language Programming for the IBM PC AT*(Prentice Hall, New York, 1986); or J. H. Crawford and P.P. Gelsinger, *Programming the 80386* (SYBEX, Alameda, CA, 1987).

[7] A copy of the final draft of the ANS Forth Standard document, X3J14 dpANS-6 can be downloaded in several different machine-readable formats, including F-PC hypertext, Microsoft Word™, or HTML, from the Web site *http://www.taygeta.com* .

[8] J.V. Noble, "Avoid decisions" *Computers in Physics* **5** #4 (1991) 386.

[9] W.H. Press, B.P. Flannery, S.A. Teukolsky and W.T. Vettering, *Numerical Recipes* (Cambridge University Press, New York, 1986) p. 35ff.

Julian Noble is Professor of Physics at the
Department of Physics
University of Virginia
P.O. Box 400714
Charlottesville, VA 22904-4714

He may be contacted at jvn@virginia.edu .

His interests are eclectic, both in and out of physics. His teaching philosophy is "no black boxes".