terse

Relief for the Forlorn Programmer

A language with hi-level look-and-feel, low-level control

JimNeiL

Spock to Captain Kirk: "Captain, you expect me to build a mnemonic memory unit using stone knives and bear skins!" Tools– where would we be without them? In the programmer's world they are the means to the end: a functioning program. Without them we would all be coding by fat-fingering a series of zeros and ones into the computer's memory through front panel switches. Over the years programming tools have advanced by leaps and bounds in all areas except one. It seems that after the invention of the macro assembler in the '60s, the assembly language programmer has been forgotten.

Most non-trivial programming projects could benefit from *some* assembly language programming. Speed is often required for device drivers, interrupt routines, highly repetitive tasks, bit manipulation and other real-time programming situations. Another reason could be the need to access special functions not directly accessible by a high-level language. Modern high speed processors expose a whole new class of problems to software solutions accessible only through low-level software control.

Long development schedules, maintenance considerations, and the challenge of locating highly skilled (and somewhat masochistic) programmers willing to work in the assembly language environment, have all but eliminated assembly language as a viable choice for the system designer. As a result, we all endure oversized, plodding programs and await the salvation of faster hardware to make them tolerable.

Jim is currently an Algorithm Designer with AuthenTec, Inc. in Melbourne Florida. Jim may be reached by phone at (407) 728-7005 or by E-mail at jim-neil@digital.net. The terse language syntax and compiler are Copyright © 1987-1995, Jim Neil, All Rights Reserved.

I have earned a good living programming in assembly language since 1972. Through the years, this work became less exciting and more repetitive. The housekeeping chores associated with this level of programming were taking their toll. The thrill was gone.

When I first programmed the Intel 8086 in October 1978, I was frustrated by the differences between 8086 assembly language and traditional assembly language. The 8086 assembler was much more like a *compiler* than an *assembler*. The concept of data typing, common in high-level languages, was foreign to my assembler experience. I had been accustomed to specifying operand types as part of the mnemonic not having the assembler derive them by examining the operands. The support of long symbol names, procedure definitions, and data structures further compounded my disorientation. Additionally, the abundance of instructions and the special purpose registers coupled with the segmented architecture made my early 8086 programming attempts intensely painful experiences.

If At First You Don't Succeed...

In desperation I invented my own traditional assembly language for the 8086 modeled after 8080 assembly language. I implemented an assembler and used it for about a year. After this experience I recognized Intel's wisdom in departing from the traditional assembler model. Programming in this traditional assembly language was no less agonizing. Worse yet, bugs were introduced as a direct result of the increased attention-to-detail burden placed upon the programmer. Not being too proud to admit a mistake, I abandoned this concept and reverted to standard 8086 assembly language.

Necessity *Is* The Mother of Invention

A few months later I was faced with another mammoth assembly language effort. The anticipation of writing that much 8086 assembly language left me in a state of despair and high-level language was out of the question due to speed/space constraints (those were the days of the 64k 4.77 MHz 8088 machines). It was imperative that I leverage my productivity. After several sleepless nights terse was conceived. The concept was for a structured symbolic language with low-level control that would help programmers be more productive. The whole key was

notation. Imagine doing calculus in English without the benefits of mathematical notation!

...Try, Try Again

terse is a computer programming language designed specifically for the Intel x86 family of microprocessors (8086/8088 through *Pentium* and beyond). It offers the programmer *complete* control over the processor (formerly only available to assembly language programmers) without the tedium and bookkeeping chores formerly associated with this type of programming. This is achieved through the utilization of a high-level language syntax without sacrificing control over the generated code. The richness of the x86 family CISC instruction set makes *terse* an extremely powerful and versatile yet extraordinarily easy to learn and use language.

A major difficulty with assembly language is that it is *vertical* in nature. What I mean by *vertical* is the single statement per line, column oriented format. *terse* is a *horizontal* free format language, allowing multiple statements per line. This allows the programmer to get a meaningful portion of the program on the screen, important when using today's full screen editors (the days of listings are gone forever). Conditional and looping statements in assembler are label and jump oriented leading to *spaghetti* code and the need to constantly invent meaningful labels. In *terse*, the conditional and looping structure is intuitive and makes the code exceedingly readable. With the proper use of indentation (a style issue, not enforced by the language) the programmer is able to get a feel for the flow of the program with only a cursory examination of the source code.

terse gives you direct access to the flags ... [and] ... all the other processor general and special purpose registers

Notation and Correlation

terse is operator rich employing prefix, postfix, and infix notation. It is a context sensitive language to which humans adapt intuitively. The selection of operators was given a considerable amount of attention. In picking each operator I insisted on having a *reason* for the choice, none was an arbitrary decision. When explaining the language I always specify the logic behind the selection the individual operators. These *reasons* assists in making the necessary mental correlation required to recall the individual operators. The commonly used operators are obvious to programmers.

Learning by Example

A complete definition of the language is beyond the scope of this article, however a brief examination of the syntax should give the reader a good idea of its structure. The spaces around some operators are a stylistic consideration and not required by the syntax of the language. Let's take a look at some examples.

Example 1(a) shows six simple *terse* statements along with their generated code. Observe the use of symbolic operators in place of the mnemonics. Notice that each statement is terminated by a semicolon. These operators should be intuitive to most programmers with the exception of the unary postfix + operator used to represent the increment (`Inc`) operation. If the *terse* statement was `ax++;`, it would generate *two* `Inc` instructions, `ax+++;` would generate three, etc. Note the register `ax` being used as the *work* register in each of these statements. In Example 1(a), I have purposely written one statement per line to show the correspondence between the *terse* and the assembly language. *terse* allows for *compound* statements to reduce the need to repeat information common to a series of instructions.

The same series of instructions could be generated with the four *terse* statements in Example 1(b). The non-assignment operations (`ax + cat - fox+;`) are performed in a strict left-to-right fashion with *no* operator precedence (i.e., `Add`, `Sub`, `Inc`). The *work* register (`ax` in this example) is repeated automatically in each generated instruction.

These four statements still contain redundant information and may be further reduced to the single *terse* statement shown in Example 1(c). Assignments are handled differently than non-assignment operations. Starting from the left, the destinations are stacked internally by the compiler until the last one (`ax` in this example) is encountered. This last destination becomes the *work* register. The last (rightmost) assignment (`ax = dog`) is performed first, followed immediately by any arithmetic/logical instructions in strict left to right order. Once the expression is fully evaluated in the *work* register, the stacked destinations are used to generate the remaining assignments, in right-to-left order using the *work* register as the source for each. This one compound *terse* statement generates the *exact same* code as the multiple terse statements shown in both Examples 1(a) and 1(b).

terse	Generated Code

(a)
`ax = dog;`
`ax + cat;`
`ax - fox;`
`ax+;`
`cow = ax;`
`elk = ax;`
	
	`Mov    ax,dog`
	`Add    ax,cat`
	`Sub    ax,fox`
	`Inc    ax`
	`Mov    cow,ax`
	`Mov    elk,ax`

(b)
`ax = dog; ax + cat - fox+;`
`cow = ax; elk = ax;`	

(c) `elk = cow = ax = dog + cat - fox+;`	

Example 1: (a) Six simple terse statements shown along with their corresponding generated code; (b) the same code is generated with four statements, using a compound arithmetic statement; (c) again the same code, this time combining compound assignment and compound arithmetic statements.

Go with the Flow

Now let's take a look at some examples of flow control. Looping is a very common occurrence in all programming languages. In assembly language, it is typically accomplished using conditional jumps (*gotos*). A trivial looping is a delay *while* loop. Example 2 demonstrates a simple delay loop implemented by loading the accumulator with 1000 and decrementing it until it becomes zero. The unary postfix − operator represents the decrement (`Dec`) operation that works like the increment operation described earlier. The <> operator is used to indicate *not equal* (greater than or less than). I didn't use `!=` because `!` is used to denote interrupt (i.e., `!21h;` generates `Int 21h`). The use of a conditional operator followed by a label (e.g., `<>Delay;`) implies a conditional jump instruction (e.g., `Jne Delay`), Again, this example is written as four separate statements to demonstrate the correspondence between the *terse* and the generated assembly language.

These statements can be re-written as shown in Example 3. Observe

terse	Generated Code

`    ax = 1000;`
`Delay:`
`    ax-;`
`    <>Delay;`
	`    Mov    ax,1000`
	`Delay:`
	`    Dec    ax`
	`    Jne    Delay`

Example 2: A simple delay loop.

terse	Generated Code

`ax = 1000;`
`{ ax-; }<>;`
	`    Mov    ax,1000`
	`?Top01:`
	`    Dec    ax`
	`    Jne    ?Top01`

Example 3: The simple delay loop, re-written using terse structured flow control constructs.

the use of braces to define the bounds of the *do-while* loop. Any number of statements may be placed inside the braces, including other nested braces. One of the real advantages of *terse* begins to come to light in this example. Notice that *terse* gives you direct access to the flags (condition codes) as defined by the individual processor operations. This, coupled with the direct control over *all* the other processor general and special purpose registers, puts *terse* in a class by itself. The compiler also takes care of the label generation housekeeping chores.

Simple *if* statements tend to be complicated to write and even more difficult to read in assembly language. *terse* supports both *if-then* and *if-then-else* statements. Example 4 is an *if* statement that sets `al` to `5` if it is *above* `5`: The statement `al - 5 ?` appears, at first glance, to be a subtraction operation. The question mark (`?`) terminator (as opposed to the usual semicolon) modifies the meaning of the statement. It is well known that an arithmetic comparison operation is simply a subtraction where the flags are set but the result is not stored. The question mark tells *terse* that you are not interested in the result, you simply wish to compare the operands. So, you can think of the question mark operator as the "just asking" operator. The use of the >> operator to denote *above* was chosen because > already denotes *greater than*. *Above* is an <u>*unsigned*</u> operation while *greater than* is a <u>*signed*</u> operation. Since unsigned operators work with potentially larger numbers, it seemed logical to use a (physically) larger operator. Also, the *greater than* operator (>) was well defined and the *above* operator was not.

Now let's look at an *if-then-else*. Example 5 sets `ah` to `1` if `ah` is zero, otherwise it is set to `0`. The `ah ?` statement sets the flags, based on a comparison of the specified register with zero, by emitting a `Test ah,ah` instruction. Though some programmers use `Or ah,ah` and others use `And ah,ah`, all three do *exactly* the same thing. Take note of the comma separating the two groups of statements delimited by braces. The == is read as *if equal* (to zero in this case), the `{` is read as *then*, the `},{` as *else* and the `};` as *endif*.

As we have seen, flow control conditionals may be placed at the *beginning* (*if*) or *end* (*while*) of a group of statements enclosed by braces. *terse* allows conditionals to be placed at *both* ends of a block creating an *if-do-while* statement. The statements in Example 6 delay `ax` times through the loop if-and-only-if `ax` is non-zero.

Some Miscellaneous Examples

The following are a few disjoint examples of some other features of the *terse* language. Example 7 shows how the *programmer* has control over the evaluation of constant sub expressions. The statement in Example 7(a) generates *two* instructions, forcing the processor to evaluate the sub expression at run time. The statement in Example 7(b) generates a single statement and evaluates the parenthetical sub expression at compile/assembly time, before the program is run. As a general rule, anything you don't want *terse* to "see" should be enclosed in parenthesis.

A common function is to "walk" a pointer through an array or list. You can see in Example 8 how the ability to pre/post increment and/or decrement index and base registers saves coding time and makes the code easier to read. The *terse* statement in Example 8(b) uses the infix binary operator ^ to denote the exclusive or (`Xor`) operation (just like in "C").

The *terse* statement in Example 9(a) is a `Push` operation. This syntax was chosen because push is a form of assignment, but the *destination* is not specified, it is implied. Example 9(b) is a `Pop` operation (a form of assignment where the *source* is not specified).

The two statements in Example 10 demonstrate compound `Push` and `Pop` statements. Observe that the `Push`es (Example 10(a)) are performed left-to-right and the `Pop`s (Example 10(b)) are performed right-to-left, forever eliminating the need to invert the order of these operations in your head.

These few examples should give you a good sense of the basic look-and-feel of the language. There is a mini-quiz below to test your *terse* reading ability. There are many operations I have not yet explained, but using your knowledge of the Intel x86 architecture, and the *terse* syntax I have explained so far, you should be able to derive most of them.

terse	Generated Code
`al - 5 ? >>`	`        Cmp   al,5`
`{ al = 5; };`	`        Jbe   ?Els02`
	`?Top02:`
	`        Mov   al,5`
	`?Els02:`

Example 4: *A simple if statement using terse structured flow control constructs. If al is above 5 then al = 5.*

terse	Generated Code
`ah ? ==`	`        Test  ah,ah`
`{ ah = 1; },`	`        Jne   ?Els03`
`{ ah = 0; };`	`?Top03:`
	`        Mov   ah,1`
	`        Jmp   ?End03`
	`?Els03:`
	`        Mov   ah,0`
	`?End03:`

Example 5: *An if-then-else using terse structured flow control constructs. If ah is zero then ah = 1 else ah = 0.*

terse	Generated Code
`ax ? <>`	`        Test  ax,ax`
`{ ax-; }<>;`	`        Je    ?Els05`
	`?Top05:`
	`        Dec   ax`
	`        Jne   ?Top05`
	`?Els05:`

Example 6: *A double ended if-while loop, using terse structured flow control constructs. If ax is non-zero then decrement ax and loop while ax is non-zero*

terse	Generated Code
(a)	
`ax = 1 + 2;`	`Mov   ax,1`
	`Add   ax,2`
(b)	
`bx = (1 + 2);`	`Mov   bx,1+2`

Example 7: *(a) A constant sub expression, evaluated at run time; (b) the sub expression is "hidden" from terse by the parenthesis and is subsequently evaluated at compile/assembly time, not run time.*

terse	Generated Code
(a)	
`dx = [si++];`	`Mov   dx,[si]`
	`Inc   si`
	`Inc   si`
(b)	
`al ^ [-bx];`	`Dec   bx`
	`Xor   al,[bx]`

Example 8: *Pre and post increment/decrement constructs. (a) post word increment; (b) pre byte decrement.*

terse	Generated Code
(a)	
`=si;`	`Push  si`
(b)	
`di=;`	`Pop   di`

Example 9: *(a) A simple Push; (b) a simple Pop. Think of them as assignments with implied operands.*

terse	Generated Code

(a) `=ax =bx =cx;`	`Push ax`
	`Push bx`
	`Push cx`

(b) `ax= bx= cx=;`	`Pop cx`
	`Pop bx`
	`Pop ax`

Example 10: (a) A compound Push statement; (b) a compound Pop statement.

A Few Comments About Comments

I have always been a firm believer in copious internal program documentation, especially in low-level programs. In designing terse, it was a requirement that commenting be a simple task. I have always been a fan of the comment-to-end-of-line style of comments. My philosophy is "If you don't have anything to say about a line of code, what is it doing in the program?" Therefore, I needed a single lowercase character that was pleasing to the eye and not requisite for the specification of any processor operation. The terse commenting character is the backslash (downhill slash). It fits all of these requirements. All characters following an unprotected \ are ignored as shown in Example 11.

Now that you have mastered the syntax, take a look at Listing One (`sieve.t`), a terse implementation of the Sieve of Eratosthenes. It generates a DOS `.COM` program that computes and displays all of the primes in the first 65536 integers. The internal documentation should make it intelligible.

eliminates most of the tedium of assembly language while delivering absolute and complete control

Complete Compatibility

Understanding the realities of the world of software development, compatibility with all of the many existing development environments was a primary concern. The compiler needed to work with existing tools, no matter who's assembler, compilers and linkers were being used on a project. The existence of multiple object file formats and the complexity of generating compatible object code lead me to an obvious decision– don't generate object code at all! Instead I chose to generate assembly language and use the assembler as a common code generator, insuring compatibility with any object file format. This allows the programmer to exploit any and all peculiarities of an individual vendors implementation.

The choice to generate assembly language however meant increased overhead at compile time. The user would have to compile the terse and then assemble the output of the compiler. I hate waiting for computers to do *anything* so I felt it was unacceptable to create any significant additional overhead. Again there was an obvious solution to this dilemma– construct a compiler that was so fast that the additional overhead incurred would be inconsequential.

Faster Than a Speeding Bullet

terse is an *extremely* fast compiler. Compilation times are measured in

terse
`ax = 7FFFh; { ax-; }<>;    \ delay a while..`

Example 11: The terse commenting character is the backslash (downhill slash) and signifies a comment to the end of the line.

thousands of lines per second! To accomplish this, I used all of my real-time programming knowledge and experience. I wrote the compiler as if I was writing a real-time program. Both the lexical and syntax analyzers are fully table driven. The lexical analyzer makes extensive use of jump tables and special hardware features of the Intel x86 architecture, like the `Xlat` and `Lodsb` instructions, to eke every cycle out of the processor. The tables for the syntax analyzer are an exhaustive enumeration of the language. The terse compiler is implemented in assembler and was the last x86 assembly language program I ever wrote. All of this, combined with huge I/O buffers, make the incremental overhead incurred an insignificant price to pay.

Pass the Buck

Another benefit of emitting assembler is the capacity for the user to intermix assembly language statements along with terse statements. This is particularly useful for accessing the pseudo-ops which vary from one implementation to the next. Any statement not recognized to be a valid terse statement is passed through to the assembler. This also means that if you feel more comfortable using standard assembler syntax for some instructions you may, with the additional benefit of multiple statements per line.

Portability or VHS vs. Beta

Since terse is Intel x86 specific, you might consider it to be a non-portable language. However, considering that the overwhelming majority of the PCs in the world are Intel x86 based, this isn't much of a penalty. More importantly, since terse has a hi-level look and feel, when it comes time to down-code a routine for efficiency, terse makes the port to low-level simple because of its familiar operators and structure. You get to think the way you are used to thinking, in a horizontal, structured, symbolic way.

Conclusion

Ever since I began using terse the joy I once had for programming has returned. Programming in terse reminds me of sailing, it's like getting something for nothing. Its symbolic notation makes it ideal for learning low-level programming and for non-English speaking programmers too. (What does `Mov` mean in Japanese?) It eliminates most of the tedium of assembly language while delivering absolute and complete control of the processor to the programmer. terse has been used in medical, military and industrial applications since 1988. The compiler is fully tested and the language has proven itself in the real world. Learn terse today and start enjoying the benefits of the first *real* improvement in low-level programming since the invention of the macro assembler.

DDJ

terse Mini Quiz

	terse	Hints
1.	`ax >> 1; dx > 1;` `bx << 2; cx < 3;`	These are *NOT* comparison operations. The left ones are somewhat like "C".
2.	`*ah; **dx;` `/cl; //bx;`	Remember larger operators work on *unsigned* values and think about x86 *implied* registers.
3.	`ax - bx; -ax;`	These should be obvious.
4.	`ax ^ bx; ^ax;`	These are the logical equivalent of the example above.
5.	`ax + 10; dx ++ 0;`	The ++ is *NOT* an unsigned operation.
6.	`'  cat;` `"  dog =1;` `"" ptr =dog;` `'  fox[8];`	These are *real* tough. They declare variables, allocate storage and initialize memory. *Now* can you figure them out?
7.	`{ !-; }.;`	If you have read the article you know "`!`" means interrupt and `.lab` is a `Jmp lab`. This will intentionally hang the computer!
8.	`cx = 1000; {}-.;`	Yet another delay, this one uses a register specific instruction.
9.	`al & bl; &dx;`	The first one is easy. To solve the other one think of `And`ing something with *nothing*.
10.	`=.Sub1; .=;`	If "`.`" means `Jmp` and "`=ax`" means `Push`, "`=.`" must mean...? The other one occurs at the end of `Sub1`.

Answers to the *terse* Mini Quiz

Intel x86 Assembler		Explanations

1.	`Ror  ax,1`
	`Shr  dx,1`
	`Rol  bx,2`
	`Shl  cx,3`
		The ">>" and "<<" operators are rotates, the ">" and "<" operators are shifts. Why? Because rotates do a *bit* more work than shifts!

2.	`Imul ah`
	`Mul  dx`
	`Idiv cl`
	`Div  bx`
		The "`*`" is an integer (signed) multiply and the "`**`" is unsigned. The x86 uses assumed dedicated registers for the destinations so you only need to (get to) specify the source. The second pair are integer and unsigned divisions.

3.	`Sub  ax,bx`
	`Neg  ax`
		These were too easy.

4.	`Xor  ax,bx`
	`Not  ax`
		The `Xor` was explained in the article. The "^" was used for `Not` because exclusive or can be thought of as a *logical* subtract.

5.	`Add  ax,10`
	`Adc  dx,0`
		The "++" is used for add with carry because it does a *bit* more than an add!

6.	`cat db   ?`
	
	`dog dw   1`
	
	`ptr dd   dog`
	
	`fox db   8 Dup(?)`
		A leading single quote is a declare byte (`db`) statement. A leading double quote is a declare word (`dw`) (double byte) statement. Two leading double quotes is a declare double word (`dd`) (four byte) statement. The brackets allow for array declarations.

7.	`lab:`
	`    Cli;`
	`    Jmp  lab`
		The "`!-`" means interrupts off. The "`}.`" can be thought of as *do-forever*.

8.	`    Mov  cx,1000`
	`lab:`
	`    Loop lab`
		The "`-.`" can be read as *decrement-and-jump*, better known as the `Loop` operation on the Intel x86.

9.	`And  al,bl`
	`Xor  dx,dx`
		The first one was easy, no? The other one solves the problem of the missing *clear* instruction on the Intel x86.

10.	`Call Sub1`
	`Ret`
		Think of "`=.`" as *push-jump* and "`.=`" as *jump-from-stack*.

Listing One

```
Name Sieve;                             \ name of program module in .OBJ file.
Title "Sieve of Eratosthenes";          \ title of program for listing file.
.186;                                   \ allow Intel 186 & above instructions.

main Group code,data;                   \ declare Group main as 2 segments.
Assume cs:main,ss:main,ds:main,es:main; \ all Segment regs point to main Group.

O   Equ <Offset main:>;                 \ define O as Offset operator.
H   Equ <256 *>;                        \ define H as High operator.
L   Equ <+>;                            \ define L as Low operator.

code Segment;                           \ define code Segment.
Org 0100h;                              \ all .COM programs start at 0100h.

data Segment;                           \ define data Segment.
        ' First2 ='1', =13, =10,        \ Initial 2 primes message text...
                 ='2', =13, =10, =24h;  \ followed by a $ for DOS.
        ' Primes =" primes.",           \ declare message text...
                  crlf =13, =10, =24h;  \ followed by CR, LF and $ for DOS.
EOP     Label Byte;                     \ define End Of Program.
data EndS;                              \ close data segment, goes after code.

\\\\\\\\
Sieve: \
\\\\\\\\
\
\   Computes and displays all of the primes between 0 and 65536 using the
\   Eratosthenes' Sieve method.  Note that the first 2 primes (1 and 2)
\   are handled as a special case.
\
\   Printing takes ~100 times longer than computing the primes. For testing
\   speed, comment out the lines that have the comments beginning with a \*
\
\   Register Usage:
\       ax = scratch.
\       bx = count, counts number of primes.
\       cx = scratch.
\       dx = prime, the current prime we are working with.
\       si = i, an index into the flags array.
\       di = k.
\

        dx = O(First2); ah = 9; !21h;   \* print first 2 primes using DOS.
        sp = O(EOP + 512);              \ set up 256 word stack at end of prog.
        bx = sp + 15 > 4;               \ bx = number of paragraphs we use.
        es = ds = ax = cs + bx;         \ setup ds,es to free space past stack.

        cx = 32768; ax = (-1); &di;     \ cx = number, ax = value, di = offset.
        +; <> ** =;                     \ auto-inc, clear full 64K flags array.

        bx = 2; &si; &ch;               \ count = 2 (for 1 & 2), i = 0, ch = 0.
        {                               \ do...
          cl = [si]; ?<>                \ if flags[i] is non-zero...
          {                             \ then...
            dx = si + si + 3; <<1;      \ prime = i * 2 + 3, break if done...
            ax = dx; =.PrintNum;        \* print prime using PrintNum.
            ax = H(14) L(13); !10h;     \* print CR using BIOS.
            ax = H(14) L(10); !10h;     \* print LF using BIOS.
            di = si + dx; >>            \ k = i + prime, if & while k << limit,
            { [di] = ch; di + dx; }>>;  \ do flags[k] = 0, k = k + prime.
            bx+;                        \ count = count + 1;
          };                            \ endif flags[i] is non-zero.
          si+;                          \ i = i + 1.
        }.;                             \ loop forever-- break gets us out.

        es = ds = ax = cs;              \ restore ds and es.
        ax = bx; =.PrintNum;            \ print ax in decimal to screen.
        dx = O(Primes); ah = 9; !21h;   \ print " primes." using DOS.
        !20h;                           \ return to DOS.

\\\\\\\\\\
PrintNum \
\\\\\\\\\\
\
\   PrintNum prints the unsigned number in ax to the screen in decimal.
\
\   Entry Conditions:
\       ax = unsigned number to convert.
\
\   Exit Conditions:
\       all registers are preserved.
\

Proc Near;                              \ procedure to convert & print numbs.
        =ax =bx =cx =dx;                \ save registers on stack.
        bx = 10; &cx;                   \ bx = base (10), cx = 0 (counts digs).
        {                               \ do...
          &dx; //bx;                    \ dxax = num, dx = rem, ax = quo.
          dl | '0'; =dx;                \ convert, push ASCII digit onto stack.
          cx+; ax?                      \ count digit, if any left to do...
        }<>;                            \ loop 'til done...
        bx = 15;                        \ bx = color (bright white).
        { ax=; ah = 14; !10h; }-.;      \ ax = pop dig, print, loop 'til done.
        ax= bx= cx= dx=;                \ restore registers from stack.
        .=;                             \ return..............................
PrintNum EndP;                          \ end proc to convert & print numbs.

code EndS;                              \ close code Segment, goes before data.
End Sieve;                              \ end of source, start at Sieve.
```