Math

If you want to make games, you will likely need math routines, if you are making a math program, you need math, and if you are making a utility, you will need math. You will need math in a good number of programs, so here are some routines that might prove useful.

Fold

Table of Contents

Navigation

Welcome page
Site Policies
Contact Info
Tutorials
Appendices
Discussion Forums
Instructions Set

External Links

WikiTI
TiCalc.org
TI Basic Developer
Other Links...

Tool Box

Recent changes
List all pages
Page Tags
Wikidot Syntax
Site Manager

Members

How to join this site?
Todo list
Site members
Invite a Friend

Page tags

It seems you have no tags attached to pages. To attach a tag simply click on the *tags* button at the bottom of any page.

edit this panel

Multiplication

DE_Times_A

At 13 bytes, this code is a pretty decent balance of speed and size. It multiplies DE by A and returns a 16-bit result in HL.

```
DE_Times_A:
;Inputs:
;     DE and A are factors
;Outputs:
;     A is not changed
;     B is 0
;     C is not changed
;     DE is not changed
;     HL is the product
;Time:
;     342+6x
;
     ld b,8          ;7           7
     ld hl,0         ;10         10
       add hl,hl     ;11*8       88
       rlca          ;4*8        32
       jr nc,$+3     ;(12|18)*8  96+6x
         add hl,de   ;--         --
       djnz $-5      ;13*7+8     99
     ret             ;10         10
```

DE_Times_A_Unrolled

Unrolled routines are larger in most cases, but they can really save on speed. This is 25% faster at its slowest, 40% faster at its fastest:

```
;==========================================================
DE_Times_A:
;==========================================================
;Inputs:
;     DE and A are factors
;Outputs:
;     A is unchanged
;     BC is unchanged
;     DE is unchanged
;     HL is the product
;speed: min 199 cycles
;       max 261 cycles
;        212+6b cycles +15 if odd, -11 if non-negative
;====================================Cycles===============
;1
     ld hl,0            ;210000          10       10
     rlca                   ;07              4
     jr nc,$+5 \ ld h,d \ ld e,l  ;3002626B   12+14p
;2
     add hl,hl              ;29             --
     rlca                   ;07              4
     jr nc,$+3 \ add hl,de  ;300119      12+6b
;3
     add hl,hl              ;29             11
     rlca                   ;07              4
```

```
;4
      add hl,hl             ;29          11
      rlca                  ;07           4
      jr nc,$+3 \ add hl,de ;300119     12+6b
;5
      add hl,hl             ;29          11
      rlca                  ;07           4
      jr nc,$+3 \ add hl,de ;300119     12+6b
;6
      add hl,hl             ;29          11
      rlca                  ;07           4
      jr nc,$+3 \ add hl,de ;300119     12+6b
;7
      add hl,hl             ;29          11
      rlca                  ;07           4
      jr nc,$+3 \ add hl,de ;300119     12+6b
;8
      add hl,hl             ;29          11
      rlca                  ;07           4
      ret nc                ;D0         11-6b
      add hl,de             ;300119     12+6b
      ret                   ;C9          10
```

A_Times_DE

This routine uses another clever way of optimising for speed without unrolling. The result is slightly larger and a bit faster. The idea here is to remove leading zeros before multiplying.

```
A_Times_DE:
;211 for times 1
;331 tops
;Outputs:
;      HL is the product
;      B is 0
;      A,C,DE are not changed
;      z flag set
;
      ld hl,0
      or a
      ld b,h     ;remove this if you don't need b=0 for outpu
      ret z
      ld b,9
        rlca
        dec b
        jr nc,$-2
Loop1:
      add hl,de
Loop2:
      dec b
      ret z
      add hl,hl
      rlca
      jp c,Loop1       ;21|20
      jp Loop2
;22 bytes
```

DE_Times_BC

```
DE_Times_BC:
;Inputs:
;      DE and BC are factors
;Outputs:
;      A is 0
;      BC is not changed
;      DEHL is the product
```

```
        ld a,16
Mul_Loop_1:
          add hl,hl
          rl e \ rl d
          jr nc,$+6
            add hl,bc
            jr nc,$+3
            inc de
          dec a
          jr nz,Mul_Loop_1
        ret
```

C_Times_D

```
C_Time_D:
;Outputs:
;      A is the result
;      B is 0
     ld b,8             ;7            7
     xor a              ;4            4
       rlca             ;4*8         32
       rlc c            ;8*8         64
       jr nc,$+3        ;(12|11)     96|88
         add a,d        ;--
       djnz $-6         ;13*7+8      99
     ret                ;10          10
;304+b (b is number of bits)
;308 is average speed.
;12 bytes
```

D_Times_C

This routine returns a 16-bit value with C as the overflow.

```
;Returns a 16-bit result
;
;===========================================================
D_Times_C:
;===========================================================
;Inputs:
;      D and C are factors
;Outputs:
;      A is the product (lower 8 bits)
;      B is 0
;      C is the overflow (upper 8 bits)
;      DE, HL are not changed
;Size:  15 bytes
;Speed: 312+12z-y
;      See Speed Summary below
;===========================================================
     xor a            ;This is an optimised way to set A to ze
     ld b,8           ;Number of bits in E, so number of times
Loop:
     add a,a          ;We double A, so we shift it left. Overf
     rl c             ;Rotate overflow in and get the next bit
     jr nc,$+6        ;If it is 0, we don't need to add anythi
       add a,d        ;Since it was 1, we do A+1*D
       jr nc,$+3      ;Check if there was overflow
         inc c        ;If there was overflow, we need to incre
     djnz Loop        ;Decrements B, if it isn't zero yet, jum
     ret

;Speed Summary
;      xor a            ;  4
;      ld b,8           ;  7
;Loop:                  ;
;      add a,a          ; 32
```

```
;       jr nc,$+6       ; 96+12z-y    z=number of bits in C, y i
;         add a,d      ; --
;         jr nc,$+3    ; --
;           inc c      ; --
;       djnz Loop      ; 99
;       ret            ; 10
;
;312+12z-y
;       z is the number of bits in C
;       y is the number of overflows in the branch. This is a
;Max: 415 cycles
```

DEHL_Mul_IXBC

32-bit multiplication

```
;***********
;**  RAM  **
;***********
;This uses Self Modifying Code to get a speed boost. This m
;used from RAM.
DEHL_Mul_IXBC:
;Inputs:
;     DEHL
;     IXBC
;Outputs:
;     AF is the return address
;     IXHLDEBC is the result
;     4 bytes at TempWord1 contain the upper 32-bits of the
;     4 bytes at TempWord3 contain the value of the input st
;Comparison/Perspective:
;     At 6MHz, this can be executed at the slowest more than
;     times per second.
;     At 15MHz, this can be executed at the slowest more tha
;     1815 times per second.
;===========================================================
     ld (TempWord1),hl
     ld (TempWord2),de
     ld (TempWord3),bc
     ld (TempWord4),ix
     ld a,32
     ld bc,0
     ld d,b \ ld e,b
Mult32StackLoop:
     sla c \ rl b \ rl e \ rl d
     .db 21h                      ;ld hl,**
TempWord1:
     .dw 0
     adc hl,hl
     .db 21h                      ;ld hl,**
TempWord2:
     .dw 0
     adc hl,hl
     jr nc,OverFlowDone
       .db 21h
TempWord3:
       .dw 0
       add hl,bc
       ld b,h \ ld c,l
       .db 21h
TempWord4:
       .dw 0
       adc hl,de
       ex de,hl
       jr nc,OverFlowDone
         ld hl,TempWord1
         inc (hl) \ jr nz,OverFlowDone
         inc hl \ inc (hl) \ jr nz,OverFlowDone
```

```
OverFlowDone:
     dec a
     jr nz,Mult32StackLoop
     ld ix,(TempWord2)
     ld hl,(TempWord1)
     ret
```

DEHL_Times_A

```
;===========================================================
DEHL_Times_A:
;===========================================================
;Inputs:
;     DEHL is a 32 bit factor
;     A is an 8 bit factor
;Outputs:
;     interrupts disabled
;     BC is not changed
;     AHLDE is the 40-bit result
;     D'E' is the lower 16 bits of the input
;     H'L' is the lower 16 bits of the output
;     B' is 0
;     C' is not changed
;     A' is not changed
;===========================================================
     di
     push hl
     or a
     sbc hl,hl
     exx
     pop de
     sbc hl,hl
     ld b,8
mul32Loop:
       add hl,hl
       rl e \ rl d
       add a,a
       jr nc,$+8
         add hl,de
         exx
         adc hl,de
         inc a
         exx
       djnz mul32Loop
       push hl
       exx
       pop de
       ret
```

H_Times_E

This is the fastest and smallest *rolled* 8-bit multiplication routine here, and it returns the full 16-bit result.

```
H_Times_E:
;Inputs:
;     H,E
;Outputs:
;     HL is the product
;     D,B are 0
;     A,E,C are preserved
;Size:  12 bytes
;Speed: 311+6b, b is the number of bits set in the input HL
;       average is 335 cycles
;       max required is 359 cycles
```

```
    ld l,d      ;6A      4       4
    ld b,8      ;0608    7       7
                ;
    add hl,hl   ;29      11*8    88
    jr nc,$+3   ;3001 12*8-5b    96-5b
    add hl,de   ;19      11*b    11b
    djnz $-4    ;10FA  13*8-5    99
                ;
    ret         ;C9      10      10
```

H_Times_E (Unrolled)

```
H_Times_E:
;Inputs:
;     H,E
;Outputs:
;     HL is the product
;     D,B are 0
;     A,E,C are preserved
;Size:  38 bytes
;Speed: 198+6b+9p-7s, b is the number of bits set in the in
;   average is 226.5 cycles (108.5 cycles saved)
;   max required is 255 cycles (104 cycles saved)
    ld d,0      ;1600   7   7
    ld l,d      ;6A     4   4
    ld b,8      ;0608   7   7
          ;
    sla h   ;   8
    jr nc,$+3   ;3001  12-b
    ld l,e   ;6B    --

    add hl,hl   ;29    11
    jr nc,$+3   ;3001  12+6b
    add hl,de   ;19    --

    add hl,hl   ;29    11
    jr nc,$+3   ;3001  12+6b
    add hl,de   ;19    --

    add hl,hl   ;29    11
    jr nc,$+3   ;3001  12+6b
    add hl,de   ;19    --

    add hl,hl   ;29    11
    jr nc,$+3   ;3001  12+6b
    add hl,de   ;19    --

    add hl,hl   ;29    11
    jr nc,$+3   ;3001  12+6b
    add hl,de   ;19    --

    add hl,hl   ;29    11
    jr nc,$+3   ;3001  12+6b
    add hl,de   ;19    --

    add hl,hl   ;29   11
    ret nc      ;D0   11+15p
    add hl,de   ;19   --
    ret         ;C9   --
```

L_Squared (fast)

The following provides an optimized algorithm to square an 8-bit number, but it only returns the lower 8 bits.
See here for somewhat of an explanation.

```
;Input: L
;Output: L*L->A
;147 t-states
;36 bytes
    ld b,l
;First iteration, get the lowest 3 bits of -x^2
    sla l
    rrc b
    sbc a,a
    or l
    ld c,a
;second iteration, get the next 2 bits of -x^2
    rrc b
    sbc a,a
    xor l
    and $F8
    add a,c
    ld c,a
;third iteration, get the next 2 bits of -x^2
    sla l
    rrc b
    sbc a,a
    xor l
    and $E0
    add a,c
    ld c,a
;fourth iteration, get the eight bit of x^2
    sla l
    rrc b
    sbc a,a
    xor l
    and $80
    sub c
    ret
```

Absolute Value

Here are a handful of optimised routines for the absolute value of a number:

```
absHL:
     bit 7,h
     ret z
     xor a \ sub l \ ld l,a
     sbc a,a \ sub h \ ld h,a
     ret
absDE:
     bit 7,d
     ret z
     xor a \ sub e \ ld e,a
     sbc a,a \ sub d \ ld d,a
     ret
absBC:
     bit 7,b
     ret z
     xor a \ sub c \ ld c,a
     sbc a,a \ sub b \ ld b,a
     ret
absA:
     or a
     ret p
     neg          ;or you can use      cpl \ inc a
     ret
```

C_Div_D

This is a simple 8-bit division routine:

```
C_Div_D:
;Inputs:
;     C is the numerator
;     D is the denominator
;Outputs:
;     A is the remainder
;     B is 0
;     C is the result of C/D
;     D,E,H,L are not changed
;
     ld b,8
     xor a
       sla c
       rla
       cp d
       jr c,$+4
         inc c
         sub d
       djnz $-8
     ret
```

DE_Div_BC

This divides DE by BC, storing the result in DE, remainder in HL

```
DE_Div_BC:          ;1281-2x, x is at most 16
     ld a,16        ;7
     ld hl,0        ;10
     jp $+5         ;10
DivLoop:
       add hl,bc    ;--
       dec a        ;64
       ret z        ;86

       sla e        ;128
       rl d         ;128
       adc hl,hl    ;240
       sbc hl,bc    ;240
       jr nc,DivLoop ;23|21
       inc e        ;--
       jp DivLoop+1
```

DEHL_Div_C

This divides the 32-bit value in DEHL by C:

```
DEHL_Div_C:
;Inputs:
;     DEHL is a 32 bit value where DE is the upper 16 bits
;     C is the value to divide DEHL by
;Outputs:
;    A is the remainder
;    B is 0
;    C is not changed
;    DEHL is the result of the division
;
     ld b,32
     xor a
       add hl,hl
       rl e \ rl d
       rla
```

```
        jr c,$+4
          inc l
          sub c
        djnz $-11
      ret
```

DEHLIX_Div_C

```
DEHLIX_Div_C:
;Inputs:
;      DEHLIX is a 48 bit value where DE is the upper 16 bit
;      C is the value to divide DEHL by
;Outputs:
;     A is the remainder
;     B is 0
;     C is not changed
;     DEHLIX is the result of the division
;
     ld b,48
     xor a
       add ix,ix
       adc hl,hl
       rl e \ rl d
       rla
       cp c
       jr c,$+5
         inc ixl
         sub c
       djnz $-15
     ret
```

HL_Div_C

```
HL_Div_C:
;Inputs:
;      HL is the numerator
;      C is the denominator
;Outputs:
;      A is the remainder
;      B is 0
;      C is not changed
;      DE is not changed
;      HL is the quotient
;
       ld b,16
       xor a
         add hl,hl
         rla
         cp c
         jr c,$+4
           inc l
           sub c
         djnz $-7
       ret
```

HLDE_Div_C

```
HLDE_Div_C:
;Inputs:
;      HLDE is a 32 bit value where HL is the upper 16 bits
;      C is the value to divide HLDE by
;Outputs:
;     A is the remainder
;     B is 0
;     C is not changed
;     HLDE is the result of the division
;
```

```
    xor a
      sll e \ rl d
      adc hl,hl
      rla
      cp c
      jr c,$+4
        inc e
        sub c
      djnz $-12
    ret
```

RoundHL_Div_C

Returns the result of the division rounded to the nearest integer.

```
RoundHL_Div_C:
;Inputs:
;     HL is the numerator
;     C is the denominator
;Outputs:
;     A is twice the remainder of the unrounded value
;     B is 0
;     C is not changed
;     DE is not changed
;     HL is the rounded quotient
;     c flag set means no rounding was performed
;            reset means the value was rounded
;
       ld b,16
       xor a
         add hl,hl
         rla
         cp c
         jr c,$+4
           inc l
           sub c
         djnz $-7
       add a,a
       cp c
       jr c,$+3
         inc hl
       ret
```

Speed Optimised HL_div_10

By adding 9 bytes to the code, we save 87 cycles:
(min speed = 636 t-states)

```
DivHLby10:
;Inputs:
;     HL
;Outputs:
;     HL is the quotient
;     A is the remainder
;     DE is not changed
;     BC is 10

 ld bc,$0D0A
 xor a
 add hl,hl \ rla
 add hl,hl \ rla
 add hl,hl \ rla

 add hl,hl \ rla
 cp c
 jr c,$+4
   sub c
   inc l
```

```
ret
```

Speed Optimised EHL_Div_10

By adding 20 bytes to the routine, we actually save 301 t-states. The speed is quite fast at a minimum of 966 t-states and a max of 1002:

```
DivEHLby10:
;Inputs:
;     EHL
;Outputs:
;     EHL is the quotient
;     A is the remainder
;     D is not changed
;     BC is 10

 ld bc,$050a
 xor a
 sla e \ rla
 sla e \ rla
 sla e \ rla

 sla e \ rla
 cp c
 jr c,$+4
   sub c
   inc e
 djnz $-8

 ld b,16

 add hl,hl
 rla
 cp c
 jr c,$+4
 sub c
 inc l
 djnz $-7
 ret
```

Speed Optimised DEHL_Div_10

The minimum speed is now 1350 t-states. The cost was 15 bytes, the savings were 589 t-states

```
DivDEHLby10:
;Inputs:
;     DEHL
;Outputs:
;     DEHL is the quotient
;     A is the remainder
;     BC is 10

 ld bc,$0D0A
 xor a
 ex de,hl
 add hl,hl \ rla
 add hl,hl \ rla
 add hl,hl \ rla

 add hl,hl \ rla
 cp c
 jr c,$+4
   sub c
   inc l
 djnz $-7

 ex de,hl
```

```
add hl,hl
rla
cp c
jr c,$+4
sub c
inc l
djnz $-7
ret
```

A_Div_C (small)

This routine should only be used when C is expected to be greater than 16. In this case, the naive way is actually the fastest and smallest way:
[code]
ld b,-1
sub c
inc b
jr nc,$-2
add a,c
[/code]
Now B is the quotient, A is the remainder. It takes at least 26 t-states and at most 346 if you ensure that c>16

E_div_10 (tiny+fast)

This is how it would appear inline, since it is so small at 10 bytes (and 81 t-states). It divides E by 10, returning the result in H :

```
e_div_10:
;returns result in H
     ld d,0
     ld h,d \ ld l,e
     add hl,hl
     add hl,de
     add hl,hl
     add hl,hl
     add hl,de
     add hl,hl
```

Square Root

RoundSqrtE

Returns the square root of E, rounded to the nearest integer:

```
;==========================================================
sqrtE:
;==========================================================
;Input:
;     E is the value to find the square root of
;Outputs:
;     A is E-D^2
;     B is 0
;     D is the rounded result
;     E is not changed
;     HL is not changed
;Destroys:
;     C
;
        xor a               ;1       4          4
        ld d,a              ;1       4          4
        ld c,a              ;1       4          4
        ld b,4              ;2       7          7
```

```
        rlc d               ;2      8         32
        ld c,d              ;1      4         16
        scf                 ;1      4         16
        rl c                ;2      8         32

        rlc e               ;2      8         32
        rla                 ;1      4         16
        rlc e               ;2      8         32
        rla                 ;1      4         16

        cp c                ;1      4         16
        jr c,$+4            ;4     12|15      48+3x
          inc d             ;--    --         --
          sub c             ;--    --         --
        djnz sqrtELoop      ;2     13|8       47
        cp d                ;1      4          4
        jr c,$+3            ;3     12|11     12|11
          inc d             ;--    --         --
        ret                 ;1     10         10
;==========================================================
;Size  : 29 bytes
;Speed : 347+3x cycles plus 1 if rounded down
;   x is the number of set bits in the result.
;==========================================================
```

SqrtE

This returns the square root of E (rounded down).

```
;==========================================================
sqrtE:
;==========================================================
;Input:
;     E is the value to find the square root of
;Outputs:
;     A is E-D^2
;     B is 0
;     D is the result
;     E is not changed
;     HL is not changed
;Destroys:
;     C=2D+1 if D is even, 2D-1 if D is odd

        xor a               ;1      4          4
        ld d,a              ;1      4          4
        ld c,a              ;1      4          4
        ld b,4              ;2      7          7
sqrtELoop:
        rlc d               ;2      8         32
        ld c,d              ;1      4         16
        scf                 ;1      4         16
        rl c                ;2      8         32

        rlc e               ;2      8         32
        rla                 ;1      4         16
        rlc e               ;2      8         32
        rla                 ;1      4         16

        cp c                ;1      4         16
        jr c,$+4            ;4     12|15      48+3x
          inc d             ;--    --         --
          sub c             ;--    --         --
        djnz sqrtELoop      ;2     13|8       47
        ret                 ;1     10         10
;==========================================================
;Size  : 25 bytes
;Speed : 332+3x cycles
;   x is the number of set bits in the result. This will no
```

```
;   into perspective, under the slowest conditions (4 set b
;   in the result at 6MHz), this can execute over 18000 tim
;   in a second.
;==========================================================
```

SqrtHL

This returns the square root of HL (rounded down). It is faster than division, interestingly:

```
SqrtHL4:
;39 bytes
;Inputs:
;      HL
;Outputs:
;      BC is the remainder
;      D is not changed
;      E is the square root
;      H is 0
;Destroys:
;      A
;      L is a value of either {0,1,4,5}
;        every bit except 0 and 2 are always zero

     ld bc,0800h   ;3  10      ;10
     ld e,c        ;1  4       ;4
     xor a         ;1  4       ;4
SHL4Loop:          ;           ;
     add hl,hl     ;1  11      ;88
     rl c          ;2  8       ;64
     adc hl,hl     ;2  15      ;120
     rl c          ;2  8       ;64
     jr nc,$+4     ;2  7|12    ;96+3y   ;y is the number of
     set 0,l       ;2  8       ;--
     ld a,e        ;1  4       ;32
     add a,a       ;1  4       ;32
     ld e,a        ;1  4       ;32
     add a,a       ;1  4       ;32
     bit 0,l       ;2  8       ;64
     jr nz,$+5     ;2  7|12    ;144-6y
     sub c         ;1  4       ;32
     jr nc,$+7     ;2  7|12    ;96+15x  ;number of bits in
         ld a,c    ;1  4       ;
         sub e     ;1  4       ;
         inc e     ;1  4       ;
         sub e     ;1  4       ;
         ld c,a    ;1  4       ;
     djnz SHL4Loop ;2  13|8    ;99
     bit 0,l       ;2  8       ;8
     ret z         ;1  11|19   ;11+8z
     inc b         ;1          ;
     ret           ;1          ;
;1036+15x-3y+8z
;x is the number of set bits in the result
;y is the number of overflows (max is 2)
;z is 1 if 'b' is returned as 1
;max is 1154 cycles
;min is 1032 cycles
```

SqrtL

This returns the square root of L, rounded down:

```
SqrtL:
;Inputs:
;      L is the value to find the square root of
```

```
;       C is the result
;       B,L are 0
;      DE is not changed
;       H is how far away it is from the next smallest perfe
;       L is 0
;       z flag set if it was a perfect square
;Destroyed:
;       A
     ld bc,400h        ; 10    10
     ld h,c            ; 4      4
sqrt8Loop:             ;
     add hl,hl         ;11     44
     add hl,hl         ;11     44
     rl c              ; 8     32
     ld a,c            ; 4     16
     rla               ; 4     16
     sub a,h           ; 4     16
     jr nc,$+5         ;12|19  48+7x
       inc c
       cpl
       ld h,a
     djnz sqrt8Loop    ;13|8   47
     ret               ;10     10
;287+7x
;19 bytes
```

ConvFPAtHL

This converts a floating point number pointed to by HL to a 16 bit-value. This is like bcall(_ConvOP1) without the limit of 9999 and a bit more flexible (since the number doesn't need to be at HL):

```
ConvDecAtHL:
;Inputs:
;      HL points to the FP number to convert
;Outputs:
;       A is the 8-bit result
;       B is 0
;      DE is the 16-bit result
;      HL is incremented by 9
;      c flag reset
;      z flag reset
;Destroys:
;      C
     ld a,(hl)
     ld b,a
     inc hl
     ld a,(hl)
     ld de,8
     add hl,de
     rla \ jr c,$+6
       ld b,d
       ld e,d
       ld a,e
       ret
     push hl
     ccf
     sbc hl,de
     ld e,d
     rra \ and 0Fh
     push bc
     ld b,a \ inc b
       inc hl
       ld c,(hl)
       call ConvDecSubLoop
```

```
        call ConvDecSubLoop
        djnz $-11
        pop af
        pop hl
        or b \ ld a,e
        ret p
        xor a
        sub e
        ld e,a
        sbc a,a
        sub d
        ld d,a
        ld a,e
        ret
ConvDecSubLoop:
        push hl
        ld h,d \ ld l,e
        add hl,hl
        add hl,hl
        add hl,de
        add hl,hl
        ex de,hl
        xor a
        sla c \ rla
        sla c \ rla
        sla c \ rla
        sla c \ rla
        add a,e
        pop hl
        ld e,a
        ret nc
        inc d
        ret
```

ConvStr16

This will convert a string of base-10 digits to a 16-bit value. Useful for parsing numbers in a string:

```
;===========================================================
ConvRStr16:
;===========================================================
;Input:
;      DE points to the base 10 number string in RAM.
;Outputs:
;      HL is the 16-bit value of the number
;      DE points to the byte after the number
;      BC is HL/10
;      z flag reset (nz)
;      c flag reset (nc)
;Destroys:
;      A (actually, add 30h and you get the ending token)
;Size:  23 bytes
;Speed: 104n+42+11c
;        n is the number of digits
;        c is at most n-2
;        at most 595 cycles for any 16-bit decimal value
;===========================================================
     ld hl,0          ;  10 : 210000
ConvLoop:             ;
     ld a,(de)        ;   7 : 1A
     sub 30h          ;   7 : D630
     cp 10            ;   7 : FE0A
     ret nc           ;5|11 : D0
     inc de           ;   6 : 13
                      ;
     ld b,h           ;   4 : 44
```

```
    add hl,hl          ;  11 : 29
    add hl,hl          ;  11 : 29
    add hl,bc          ;  11 : 09
    add hl,hl          ;  11 : 29
                       ;
    add a,l            ;   4 : 85
    ld l,a             ;   4 : 6F
    jr nc,ConvLoop     ;12|23: 30EE
    inc h              ; --- : 24
    jr ConvLoop        ; --- : 18EB
```

GCDHL_BC

This computes the Greatest Common Divisor of HL and BC:

```
GCDHL_BC:
;Inputs:
;     HL,BC
;Outputs:
;     A is 0
;     BC,DE are both the GCD
;     HL is 0
     ld a,16
     ld de,0
     add hl,hl
     ex de,hl
     adc hl,hl
     or a
     sbc hl,bc
     jr c,$+3
     add hl,bc
     ex de,hl
     dec a
     jr nz,GCDHL_BC+5
     ld h,b
     ld l,c
     ld b,d
     ld c,e
     ld a,d \ or e
     jr nz,GCDHL_BC
     ret
```

Modulus

L_mod_3

Computes L mod 3 (essentially, the remainder of L after division by 3):
L_mod_3:

```
;Outputs:
;     HL is preserved
;     A is the remainder
;     destroys DE,BC
;     z flag if divisible by 3, else nz
     ld bc,030Fh
;Now we need to add the upper and lower nibble in a
     ld a,l
     and c
     ld e,a
     ld a,l
     rlca
     rlca
     rlca
```

```
    and c

    add a,e
    sub c
    jr nc,$+3
    add a,c
;add the lower half nibbles    ;at this point, we have l_mo

    ld d,a
    sra d
    sra d
    and b
    add a,d
    sub b
    ret nc
    add a,b
    ret
;at most 117 cycles, at least 108, 28 bytes
```

HL_mod_3

```
HL_mod_3:
;Outputs:
;      Preserves HL
;      A is the remainder
;      destroys DE,BC
;      z flag if divisible by 3, else nz
    ld bc,030Fh
    ld a,h
    add a,l
    sbc a,0    ;conditional decrement
;Now we need to add the upper and lower nibble in a
    ld d,a
    and c
    ld e,a
    ld a,d
    rlca
    rlca
    rlca
    rlca
    and c

    add a,e
    sub c
    jr nc,$+3
    add a,c
;add the lower half nibbles

    ld d,a
    sra d
    sra d
    and b
    add a,d
    sub b
    ret nc
    add a,b
    ret
;at most 132 cycles, at least 123
```

DEHL_mod_3

Same as HLDE_mod_3

HLDE_mod_3

```
DEHL_mod_3:
HLDE_mod_3:
```

```
;      A is the remainder
;      destroys DE,BC
;      z flag if divisible by 3, else nz
     ld bc,030Fh
     add hl,de
     jr nc,$+3
     dec hl
     ld a,h
     add a,l
     sbc a,0   ;conditional decrement
;Now we need to add the upper and lower nibble in a
     ld d,a
     and c
     ld e,a
     ld a,d
     rlca
     rlca
     rlca
     rlca
     and c

     add a,e
     sub c
     jr nc,$+3
     add a,c
;add the lower half nibbles

     ld d,a
     sra d
     sra d
     and b
     add a,d
     sub b
     ret nc
     add a,b
     ret
;at most 156 cycles, at least 146
```

A_mod_10:

This is not a typical method used, but it is small and fast at 196 to 201 t-states, 12 bytes

```
     ld bc,05A0h
Loop:
     sub c
     jr nc,$+3
       add a,c
     srl c
     djnz Loop
     ret
```

PseudoRandByte_0

This is one of many variations of PRNGs. This routine is not particularly useful for many games, but is fairly useful for shuffling a deck of cards. It uses SMC, but that can be fixed by defining randSeed elsewhere and using ld a,(randSeed) at the beginning.

```
PseudoRandByte:
;f(n+1)=13f(n)+83
;97 cycles
     .db 3Eh     ;start of ld a,*
randSeed:
     .db 0
     ld c,a
```

```
        add a,c
        add a,a
        add a,a
        add a,c
        add a,83
        ld (randSeed),a
        ret
```

PseudoRandWord_0:

Similar to the PseudoRandByte_0, this generates a a sequence of pseudo-random values that has a cycle of 65536 (so it will hit every single number):

```
PseudoRandWord:
;f(n+1)=241f(n)+257   ;65536
;181 cycles, add 17 if called
;Outputs:
;     BC was the previous pseudorandom value
;     HL is the next pseudorandom value
;Notes:
;     You can also use B,C,H,L as pseudorandom 8-bit values
;     this will generate all 8-bit values
     .db 21h     ;start of ld hl,**
randSeed:
     .dw 0
     ld c,l
     ld b,h
     add hl,hl
     add hl,bc
     add hl,hl
     add hl,bc
     add hl,hl
     add hl,bc
     add hl,hl
     add hl,hl
     add hl,hl
     add hl,hl
     add hl,bc
     inc h
     inc hl
     ld (randSeed),hl
     ret
```

Fixed Point Math

Fixed Point numbers are similar to Floating Point numbers in that they give the user a way to work with non-integers. For some terminology, an 8.8 Fixed Point number is 16 bits where the upper 8 bits is the integer part, the lower 8 bits is the fractional part. Both Floating Point and Fixed Point are abbreviated 'FP', but one can tell if Fixed Point is being referred to by context. The way one would interpret an 8.8 FP number would be to take the upper 8 bits as the integer part and divide the lower 8-bits by 256 (2[sup]8[/sup]) so if HL is an 8.8 FP number that is $1337, then its value is 19+55/256 = 19.21484375. In most cases, integers are enough for working in Z80 Assembly, but if that doesn't work, you will rarely need more than 16.16 FP precision (which is 32 bits in all).
FP algorithms are generally pretty similar to their integer counterparts, so it isn't too difficult to convert.

FPLog88

accurate. In the very worst case, it is off by 2/256, but on average, it is off by less than 1/256 (the smallest unit for an 8.8 FP number).

```
FPLog88:
;Input:
;     HL is the 8.8 Fixed Point input. H is the integer par
;Output:
;     HL is the natural log of the input, in 8.8 Fixed Poin
     ld a,h
     or l
     dec hl
     ret z
     inc hl
     push hl
     ld b,15
     add hl,hl
     jr c,$+4
     djnz $-3
     ld a,b
     sub 8
     jr nc,$+4
     neg
     ld b,a
     pop hl
     push af
     jr nz,lnx
     jr nc,$+7
     add hl,hl
     djnz $-1
     jr lnx
     sra h
     rr l
     djnz $-4
lnx:
     dec h          ;subtract 1 so that we are doing ln((x-1)
     push hl        ;save for later
     add hl,hl      ;we are doing the 4x/(4+4x) part
     add hl,hl
     ld d,h
     ld e,l
     inc h
     inc h
     inc h
     inc h
     call FPDE_Div_HL  ;preserves DE, returns AHL as the 16
     pop de         ;DE is now x instead of 4x
     inc h          ;now we are doing x/(3+Ans)
     inc h
     inc h
     call FPDE_Div_HL
     inc h          ;now we are doing x/(2+Ans)
     inc h
     call FPDE_Div_HL
     inc h          ;now we are doing x/(1+Ans)
     call FPDE_Div_HL  ;now it is computed to pretty decent
     pop af         ;the power of 2 that we divided the initi
     ret z          ;if it was 0, we don't need to add/subtra
     ld b,a
     jr c,SubtLn2
     push hl
     xor a
     ld de,$B172    ;this is approximately ln(2) in 0.16 FP f
     ld h,a
     ld l,a
     add hl,de
     jr nc,$+3
     inc a
     djnz $-4
     pop de
```