Z80 Routines:Math:Square root

From WikiTI

Contents

Size Optimization

This version is size optimized, it compares every perfect square against HL until a square that is larger is found. Obviously slower, but does get the job done in only 12 bytes.

```
;-------------------------------
;Square Root
;Inputs:
;HL = number to be square rooted
;Outputs:
;A  = square root

sqrt:
   ld a,$ff
   ld de,1
sqrtloop:
   inc a
   dec e
   dec de
   add hl,de
   jr c,sqrtloop
   ret
```

Speed Optimization

This version is an unrolled version of a square root algorithm taking at most 555 t-states. The downside is that it takes 111 bytes, but if speed is a priority, this is probably the best option. Most of the unrolled iterations are specially optimised for their location in the algorithm.

```
;-------------------------------
;Square Root
;Inputs:
;HL = number to be square rooted
;Outputs:
;A  = square root
;111 bytes
;555 t-states worst case
SqrtHL:
;zero some registers
```

```
        xor a
        ld c,a
        ld d,a

;move the LSB of the input into E for later use, then shift the LSB into L and load H with 0.
;H will be a carry register, where the bits in L are rotated in
        ld e,l
        ld l,h
        ld h,c

;Iteration 1 is optimised
; C is treated as the accumulator
        add hl,hl
        add hl,hl
        sub h
        jr nc,$+5
        inc c
        cpl
        ld h,a

;Iteration 2
; rotate in 2 more bits from the MSB of the input into H
        add hl,hl
        add hl,hl
; shift the accumulator
        rl c
        ld a,c
        rla
; A is now double the shifted accumulator
        sub h
; doubles as a comparison of the carry register (H) to double the accumulator
        jr nc,$+5
; If the carry is > 2*accumulator, the bit in the accumulator needs to be 1:
        inc c
; We need to perform H-(2C+1), but A=2C-H.
; We could do NEG to get A=H-2C, then DEC A, but NEG = CPL \ INC A
; NEG \ DEC A  =  CPL \ INC A \ DEC A
; So just use CPL, saving 8 t-states, 1 byte
        cpl
        ld h,a

;Iteration 3
        add hl,hl
        add hl,hl
        rl c
        ld a,c
        rla
        sub h
        jr nc,$+5
        inc c
        cpl
        ld h,a

;Iteration 4
        add hl,hl
        add hl,hl
        rl c
        ld a,c
        rla
        sub h
        jr nc,$+5
        inc c
        cpl
        ld h,a

;L is 0, H is the current carry
;E is the lower 8 bits
; Load the next set of bits (LSB of input) into L so that they can be rotated into H
        ld l,e

;Iteration 5
        add hl,hl
        add hl,hl
        rl c
        ld a,c
        rla
        sub h
        jr nc,$+5
        inc c
        cpl
        ld h,a

;Iteration 6
        add hl,hl
        add hl,hl
        rl c
        ld a,c
        rla
        sub h
        jr nc,$+5
        inc c
        cpl
        ld h,a

;Iteration 7
; Now we need to start worrying about 8 bit overflow.
; In particular, the carry register, H should be ideally 9 bits for this iteration, 10 for the last.
; The accumulator, C, is 8 bits, but we need to compare H to 2*C, and 2*C is up to 9 bits on the last iteration.
;l has 4 more bits to rotate into h

        sla c \ ld a,c \ add a,a
        add hl,hl
        add hl,hl
        jr nc,$+6
        sub h \ jp $+6
        sub h
        jr nc,$+5
        inc c
        cpl
        ld h,a

;Iteration 8
; A lot of fancy stuff here
; D is 0, from way back at the beginning
; now I put H->E so that DE can hold the potentially 10-bit number
; Now C->A, L->H
; H thus has the last two bits of the input that need to be rotated into DE
; L has the value of the accumualtor which needs to be multiplied by 4 for a comparison to DE
; So 2 shifts of HL into DE results in DE holding the carry, HL holding 4*accumulated result!
        ld e,h
        ld h,l
        ld l,c
        ld a,l
        add hl,hl \ rl e \ rl d
        add hl,hl \ rl e \ rl d
        sbc hl,de
;the c flag now has the state of the last bit of the result, HL does not need to be restored.
        rla
        ret
```

Balanced Optimization

This version is a balance between speed and size. It also uses the high school method and runs under 1200 tstates. It only costs 41 bytes.

```
;-------------------------------
;Square Root
;Inputs:
;DE = number to be square rooted
;Outputs:
;A  = square root

Sqrt:
    ld hl,0
    ld c,l
    ld b,h
    ld a,8
Sqrtloop:
    sla e
    rl d
    adc hl,hl
    sla e
    rl d
    adc hl,hl
    scf                 ;Can be optimised
    rl c                ;with SL1 instruction
    rl b
    sbc hl,bc
    jr nc,Sqrtaddbit
    add hl,bc
    dec c
Sqrtaddbit:
    inc c
    res 0,c
    dec a
    jr nz,Sqrtloop
    ld a,c
    rr b
    rra
    ret
```

Presumably the best

This code was found on z80 bits and has the advantage of being both faster than all three versions above and smaller than the last two (it runs in under 720 T-states (under 640 if fully unrolled) and takes a mere 29 bytes). On the other hand it takes a somewhat unconventionnal input... It computes the square root of the 16bit number formed by la and places the result in d.

```
sqrt_la:
        ld      de, 0040h       ; 40h appends "01" to D
        ld      h, d

        ld      b, 7

        ; need to clear the carry beforehand
        or      a

_loop:
        sbc     hl, de
        jr      nc, $+3
        add     hl, de
        ccf
        rl      d
        rla
        adc     hl, hl
        rla
        adc     hl, hl

        djnz    _loop

        sbc     hl, de          ; optimised last iteration
        ccf
        rl      d

        ret
```

Other Options

A binary search of a square table would yield much better best case scenarios and the worst case scenarios would be similar to the high school method. However this would also require 512 byte table making it significantly larger than the other routines. Of course the table could also serve as a rapid squaring method.

Credits and Contributions

- **James Montelongo**
- **Milos "baze" Bazelides** (or possibly one of the contributor of z80bits (http://baze.au.com/misc/z80bits.html))