Square Root calculation

Imagine you wanna have a square root:

```
R = sqrt(N)
```

That obviously gives you:

```
R^2 = N
```

Since we wanna establish a convergent algo, we simply use this during calculation:

```
R(n)^2 <= N
```

R(0) will be 0. To get closer to N we add a third value D to R as long as that formula is true. Since we work with binary computers, this D will be of the kind 2^x (128, 64, 32 etc).

Ok assuming we wanna have the root from a 16 bit number and since the output of a sqrt of max 65535 (highest 16 bit input) is maximum 255.998 we start with a D of 128 and LSR it down to 1. On each iteration we try (R+D)^2 ⇐ N and if that formula is true, we have R=R+D. If not, R stays unmodified. The resulting algo looks like this:

```
R = 0
D = 128
while (D >= 1)
{
    temp = R+D
    if (temp*temp <= N) then R=temp
    D = D/2
}
```

As you see this algo is very simple and apart from the temp*temp it does not involve any time consuming operation. The D/2 is ofcourse just a single LSR.

Ok but we are not satisfied. One single integer multiplication is too much for our little 6510 so we have to get rid of it.

Ok now we are not satisfied with the mul but how to get rid of it? First you have to see that (R+D)^2 is a simple binomical formula so you can do this:

```
(R+D)^2 = (R+D)*(R+D) = R*R + 2*R*D + D*D
```

Still this doesn't look THAT promising but let's try more formula changing:

```
R*R + 2*R*D + D*D = R*R + D*(2*R + D)
```

Ok what do we have now? 2*R is easy since it's just a simple ASL in assembler. A multiplication with D is also easy since D is always of the kind 2^x so D*something is equal to ASL something several times. But damnit, there's still that R*R biatch, let's send her to hell by introducing another variable

called M:

```
M(n) <= N - R(n)^2
```

And since R(0) = 0:

```
M(0) = N
```

As you see M(n) is just the difference from N and “current N during algo” which gives us a new way to decide if D should be added to R or not. The only thing we need now is a way to calculate M(n) without calculating R(n)^2. This is a bit messy but it works with these given formulas:

```
N(n) <= (R(n)+D(n))^2
```

Given the easy formula transformation from above we have:

```
N(n) <= R(n)^2 + D(n)*(2*R(n)+D(n))
```

Just substract R(n)^2 from both sides and get:

```
N(n) - R(n)^2 <= D(n)*(2*R(n)+D(n))
```

Now look a few lines above at the M(n) formula. Do you notice something? Exactly, there we have the front part of this formula too so it is proven that:

```
M(n) <= D(n)*(2*R(n)+D(n))
```

Time for a party, that R*R is gone! But how do we calculate those M(n)? Quite simple actually:

```
M(n+1) = M(n) - D(n)*(2*R(n)+D(n))
```

So we got a completely new algo:

```
M = N
D = 128
while (D >= 1)
{
    T = D*(2*R+D)
    if (T <= M) then M=M-T : R = R+D
    D = D/2
}
```

Yeeehaw! R*R is gone, and 2*R is easy (ASL) and D*whatever is easy too (ASL multiple times). To make this clear I change the muls into ASL's:

```
M = N
D = 128
for n = 7 to 0
{
    T = (R+R+D) ASL n
```

```
    if (T <= M) then M=M-T : R = R+D
    D = D LSR 1
}
```

Wicked! BUT: We are not satisfied because of n-times ASL :)

Ok I stopped at the point where the multiplication was gone but a ASL with varying shift count appeared. That is no problem for big CPU's like 680×0 or x86, but our poor 6510 doesn't like this so that “ASL n” has to leave.

To do that, we don't investigate the math any further but take a look at how the variables behave during iteration of this algo:

```
R = 0
M = N
D = 128
for n = 7 to 0
{
    T = (R+R+D) ASL n
    if (T <= M) then M=M-T : R = R+D
    D = D LSR 1
}
```

D, T, M and R now look like this:

```
D        T                 M                 R
10000000 0100000000000000 xxxxxxxxxxxxxxxx x0000000
01000000 0x0100000000000 0xxxxxxxxxxxxxxx xx000000
00100000 00xx010000000000 00xxxxxxxxxxxxx xxx00000
00010000 000xxx0100000000 000xxxxxxxxxxxx xxxx0000
00001000 0000xxxx01000000 0000xxxxxxxxxxx xxxxx000
00000100 00000xxxxx010000 00000xxxxxxxxxx xxxxxx00
00000010 000000xxxxxx0100 000000xxxxxxxxx xxxxxxx0
00000001 0000000xxxxxxx01 0000000xxxxxxxx xxxxxxxx
```

Removing the “ASL n” would completely change the behaviour of T. But also M has to change because otherwise the T ⇐ M compare and M-T math will not work. So if T is not left-shifted, M has to be right-shifted. The result looks like this:

```
D        T                 M                 R
10000000 0100000000000000 xxxxxxxxxxxxxxxx x0000000
01000000 x010000000000000 xxxxxxxxxxxxxxx0 xx000000
00100000 xx01000000000000 xxxxxxxxxxxxxx00 xxx00000
00010000 xxx0100000000000 xxxxxxxxxxxxx000 xxxx0000
00001000 xxxx010000000000 xxxxxxxxxxxx0000 xxxxx000
00000100 xxxxx01000000000 xxxxxxxxxxx00000 xxxxxx00
00000010 xxxxxx0100000000 xxxxxxxxxx000000 xxxxxxx0
00000001 xxxxxxx010000000 xxxxxxxxx0000000 xxxxxxxx
```

So now instead of shifting T we shift M, but the advantage of shifting M is that it's only 1 bit per iteration. Look at the code now:

```
R = 0
M = N
D = 128
for n = 7 to 0
{
    T = (R+R+D) ASL 7
    if (T <= M) then M=M-T : R = R+D
    M = M ASL 1
    D = D LSR 1
}
```

Much much better since ASL 7 can be replaced by a single LSR in asm code later. This code is quite useful on 6510 already, BUT ofcourse this is not everything.

Taking a slightly modified version of the last pseudo code:

```
R = 0
M = N
D = 128
for n = 7 to 0
{
    T = (R ASL 8) OR (D ASL 7)
    if (T <= M) then M=M-T : R = R OR D
    M = M ASL 1
    D = D LSR 1
}
```

T can now be calculated quite easily. The high byte is simply R OR <some stuff> and the low byte is always 0 except for the last iteration where it is 128. The shifting of D is done via a table and voila, here is some 6510 code:

```
        LDY #$00    ; R = 0
        LDX #$07
.loop
        TYA
        ORA stab-1,X
        STA THI     ; (R ASL 8) | (D ASL 7)
        LDA MHI
        CMP THI
        BCC .skip1  ; T <= M
        SBC THI
        STA MHI     ; M = M - T
        TYA
        ORA stab,x
        TAY         ; R = R OR D
.skip1
        ASL MLO
        ROL MHI     ; M = M ASL 1
        DEX
        BNE .loop
```

```
        ; last iteration

        STY THI
        LDA MLO
        CMP #$80
        LDA MHI
        SBC THI
        BCC .skip2
        INY         ; R = R OR D (D is 1 here)
.skip2
        RTS
stab:   .BYTE $01,$02,$04,$08,$10,$20,$40,$80
```

I hope I didn't make any mistakes in that routine :)

Input is MLO/MHI for N and output is Y-register for int(sqrt(N)).

Ok guys, I finally tested the routine I did up there. Like said before, it only allows 14 bit input ($0000 to $41FF to be more accurate). The reason for this is that M needs one more bit. Ok, some people might want full 16 bit so here is a fixed routine which only has 3 opcodes more:

```
        LDY #$00    ; R = 0
        LDX #$07
        CLC         ; clear bit 16 of M
.loop
        TYA
        ORA stab-1,X
        STA THI     ; (R ASL 8) | (D ASL 7)
        LDA MHI
        BCS .skip0  ; M >= 65536? then T <= M is always true
        CMP THI
        BCC .skip1  ; T <= M
.skip0
        SBC THI
        STA MHI     ; M = M - T
        TYA
        ORA stab,x
        TAY         ; R = R OR D
.skip1
        ASL MLO
        ROL MHI     ; M = M ASL 1
        DEX
        BNE .loop

        ; last iteration

        BCS .skip2
        STY THI
        LDA MLO
        CMP #$80
        LDA MHI
```

```
        SBC THI
        BCC .skip3
.skip2
        INY         ; R = R OR D (D is 1 here)
.skip3
        RTS
stab:   .BYTE $01,$02,$04,$08,$10,$20,$40,$80
```

This routine works perfectly for all values from $0000 to $FFFF.

Here's an extended incarnation of the integer sqrt routine. This one is extended so it does proper rounding. It is achieved by adding another iteration and calculate bit -1 which then can be used to determine if the fractional part is >= .5 or not.

The disadvantage of rounding is that the output has to be 9 bits now, because inputs $FF01 to $FFFF result in 256 now (and not 255). Alternatively you also might modify this routine and simply use bit -1 for further calculations. This gives you more accuracy than rounding.

```
        LDY #$00    ; R = 0
        LDX #$07
        CLC         ; clear bit 16 of M
.loop
        TYA
        ORA stab-1,X
        STA THI     ; (R ASL 8) | (D ASL 7)
        LDA MHI
        BCS .skip0  ; M >= 65536? then T <= M is always true
        CMP THI
        BCC .skip1  ; T <= M
.skip0
        SBC THI
        STA MHI     ; M = M - T
        TYA
        ORA stab,x
        TAY         ; R = R OR D
.skip1
        ASL MLO
        ROL MHI     ; M = M ASL 1
        DEX
        BNE .loop

        ; bit 0 iteration

        STY THI
        LDX MLO
        LDA MHI
        BCS .skip2
        CPX #$80
        SBC THI
        BCC .skip3
.skip2
```

```
        TXA
        SBC #$80
        TAX
        LDA MHI
        SBC THI
        STA MHI
        INY         ; R = R OR D (D is 1 here)
.skip3
        CPX #$80
        ROL MHI

        ; bit -1 iteration and rounding ( + 0.5)

        LDX #$00
        BCS .skip4
        CPY MHI
        BCS .skip5
.skip4
        INY         ; R = R + 0.5
        BNE .skip5
        INX
.skip5
        ; R in X and Y (Y is low-byte)

        RTS
stab:   .BYTE $01,$02,$04,$08,$10,$20,$40,$80
```

If you don't wanna have the rounding and prefer having bit -1, then use this final iteration:

```
        ; bit -1 iteration

        BCS .skip4
        LDX #$00
        CPY MHI
        BCS .skip5
.skip4
        LDX #$80
.skip5
        ; R in Y, X contains bit -1

        RTS
```

---

by Verz: the generic algorithm is:

```
        R= 0
        M= N
        D= 2^(p-1)
        for n= 1 to p
        {
```

```
            T= (R+R+D) ASL (p-1)
            if (T <= M) then M=M-T: R=R+D
            M= M ASL 1
            D= D LSR 1
        }
```

where *p* is the number of bits of the result; (or half the bits of the radicand, +1 if the radicand has an odd amount of bits).
p=ceil(#bits-of-radicand/2)

Incidentally the algorithm provides also the remainder via (M LSR p)
p can be greater than necessary, just producing more iterations. A value of p=16 permits to perform the calculations for 32bit numbers and smaller.

This is the code for a 32bit integer Sqrt. Provides the result and the remainder:

```
;*********************************************
;*    sqrt32
;*
;*   computes Sqrt of a 32bit number
;*********************************************
;*   by Verz - Jul2019
;*********************************************
;*
;*  input:  square, 32bit source number
;*  output: sqrt,   16bit value
;*          remnd,  17bit value
;*********************************************

sqrt32  lda #0
        sta sqrt        ; R=0
        sta sqrt+1
        sta M+4
        ;sta T+1        ; (T+1) is zero until last iteration; (T+0) is
always 0

        clc
        ldy #14         ; 15 iterations (14-->0) + last iteration
loopsq
        lda sqrt        ; (2*R+D) LSR 1; actually: R+(D LSR 1)
        ora stablo,y    ; using ORA instead of ADC is ok because the bit to
be set
        sta T+2         ;    will have not been affected yet
        lda sqrt+1
        ora stabhi,y
        sta T+3
        bcs skip0       ; takes care of large numbers; if set, M>T

        lda M+3
        cmp T+3
        bcc skip1       ; T <= M    (branch if T>M)
```

```
        bne skip0
        lda M+2
        cmp T+2
        bcc skip1
skip0   ;sec
        lda M+2         ; M=M-T
        sbc T+2
        sta M+2
        lda M+3
        sbc T+3
        sta M+3
        lda sqrt        ; R=R+D
        ora stablo+1,y
        sta sqrt
        lda sqrt+1
        ora stabhi+1,y
        sta sqrt+1
skip1
        asl M           ; M=M*2
        rol M+1
        rol M+2
        rol M+3
        dey             ; implicit: D=D/2, by the decrement of .Y
        bpl loopsq
lastiter                ; code for last iteration
        bcs skp0        ; takes care of large numbers; if set, M>T
        ; during last iteration D=1, so [(2*R+D) LSR 1] makes D the MSB of
T+1
        lda M+3
        cmp sqrt+1      ; (T+3) = sqrtHI
        bcc skp1        ; T <= M    branch if T>M
        bne skp0
        lda M+2
        cmp sqrt        ; (T+2) = sqrtLO
        bcc skp1
        bne skp0
        lda M+1
        cmp #$80        ; value of (T+1) during last iteration
        bcc skp1
skp0    ;sec
        lda M+1
        sbc #$80        ; (T+1) during last iteration
        sta M+1
        lda M+2         ; M=M-T
        sbc sqrt        ; (T+2)
        sta M+2
        lda M+3
        sbc sqrt+1      ; (T+3)
        sta M+3
        inc sqrt        ; R=R+D with D=1
skp1    asl M+1         ; M=M*2; location M+0=0
```

```
        rol M+2
        rol M+3
        rol M+4
        rts

;**** Variables and Shift table
stabhi byte 0,0,0,0,0,0,0,0
stablo BYTE $01,$02,$04,$08,$10,$20,$40,$80
       byte 0,0,0,0,0,0,0,0

square = $57    ; 5 bytes: input value; during calculation needs the 5th
byte
sqrt   = $5F    ; 2 bytes: result
remnd  = M+2    ; 2 B + 1 b: is in the high bytes of M (M LSR 16); msb is in
T+0 (the 5th byte of square)
T      = $5B    ; 4 bytes: could be 2 bytes: T+0 is always 0; T+1 is 0 until
last iteration
M      = square ; 4 bytes: over the input square
```

The algorithm is pretty fast: it has a top cycles count of around 1700, but seems to average at 1.3ms (using variables in page zero).