```
;(16 bit division), Fast divide by 10
;https://forums.atariage.com/blogs/entry/11044-16-bit-division-
fast-divide-by-10/



;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;    UNSIGNED DIVIDE BY 10 (16 BIT)
;    111 cycles (max), 96 bytes
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
TensRemaining:
    .byte 0,25,51,76,102,128,153,179,204,230
ModRemaing:
    .byte 0,6,2,8,4,0,6,2,8,4
.overflowFound:
    cmp    #4                   ;2  @74  We have overflowed,
but we can apply a shortcut.
    lda    #25                  ;2  @76  Divide by 10 will be
at least 25, and the
    bne    .finishLowTen        ;3  @79  carry is set when
higher for the next addition.                                ;
always branch
.startBigDivide:
    lda    counterHigh          ;3  @3
    sta    temp                 ;3  @6
    lsr                         ;2  @8
    adc    #13                  ;2  @10
    adc    temp                 ;3  @13
    ror                         ;2  @15
    lsr                         ;2  @17
    lsr                         ;2  @19
    adc    temp                 ;3  @22
    ror                         ;2  @24
    adc    temp                 ;3  @27
    ror                         ;2  @29
    lsr                         ;2  @31
    and    #$7C                 ;2  @33   AND'ing here...
    sta    temp                 ;3  @36   and saving result as
highTen (times 4)
    lsr                         ;2  @38
    lsr                         ;2  @40
    sta    highTen              ;3  @43
    adc    temp                 ;3  @46   highTen (times 5)
    asl                         ;2  @48   highTen (times 10)
```

```
    sbc    counterHigh           ;3  @51
    eor    #$FF                  ;2  @53
    tay                          ;2  @55   mod 10 result!
    lda    TensRemaining,Y       ;4  @59   Fill the low byte
with the tens it should
    sta    lowTen                ;3  @62   have at this point
from the high byte divide.
    lda    counterLow            ;3  @65
    adc    ModRemaing,Y          ;4  @69
    bcs    .overflowFound        ;2? @71/72
    sta    temp                  ;3  @74
    lsr                          ;2  @76
    adc    #13                   ;2  @78
    adc    temp                  ;3  @81
    ror                          ;2  @83
    lsr                          ;2  @85
    lsr                          ;2  @87
    adc    temp                  ;3  @90
    ror                          ;2  @92
    adc    temp                  ;3  @95
    ror                          ;2  @97
    lsr                          ;2  @99
    lsr                          ;2  @101
    lsr                          ;2  @103
    clc                          ;2  @105
.finishLowTen:
    adc    lowTen                ;3  @108
    sta    lowTen                ;3  @111
```

;Here is a second version of the divide by ten. It takes 126 cycles, but uses only 79 bytes. So you can choose whatever routine is best. Code for it is below.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;    UNSIGNED DIVIDE BY 10 (16 BIT)
;    126 cycles (max), 79 bytes
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
TensRemaining:
    .byte 0,25,51,76,102,128,153,179,204,230
ModRemaing:
    .byte 0,6,2,8,4,0,6,2,8,4
.startBigDivide:
```

```
    ldy    #-2                 ;2  @2   skips a branch the
first time through
    lda    counterHigh         ;3  @5
.do8bitDiv:
    sta    temp                ;3  @8
    lsr                        ;2  @10
    adc    #13                 ;2  @12
    adc    temp                ;3  @15
    ror                        ;2  @17
    lsr                        ;2  @19
    lsr                        ;2  @21
    adc    temp                ;3  @24
    ror                        ;2  @26
    adc    temp                ;3  @29
    ror                        ;2  @31
    lsr                        ;2  @33
    and    #$7C                ;2  @35   AND'ing here...
    sta    temp                ;3  @38   and saving result as
highTen (times 4)
    lsr                        ;2  @40
    lsr                        ;2  @42
    iny                        ;2  @44
    bpl    .finishLowTen       ;2? @46/47...120
    sta    highTen             ;3  @49
    adc    temp                ;3  @52   highTen (times 5)
    asl                        ;2  @54   highTen (times 10)
    sbc    counterHigh         ;3  @57
    eor    #$FF                ;2  @59
    tay                        ;2  @61   mod 10 result!
    lda    TensRemaining,Y     ;4  @65   Fill the low byte
with the tens it should
    sta    lowTen              ;3  @68   have at this point
from the high byte divide.
    lda    counterLow          ;3  @71
    adc    ModRemaing,Y        ;4  @75
    bcc    .do8bitDiv          ;2? @77/78
.overflowFound:
    cmp    #4                  ;2  @79   We have overflowed,
but we can apply a shortcut.
    lda    #25                 ;2  @81   Divide by 10 will be
at least 25, and the
                               ;         carry is set when
higher for the next addition..
finishLowTen:
    adc    lowTen              ;3  @123
```

```
    sta    lowTen               ;3  @126  routine ends at
either 87 or 126 cycles
```

```
;16-bit increment and decrement
;https://www.nesdev.org/wiki/Synthetic_instructions
;Incrementing/decrementing a 16-bit value involves first
adjusting the low byte, then adjusting the high byte if
necessary. Increment is simpler, since the high byte is
adjusted when the low byte wraps around to zero; for decrement,
the high byte is adjusted when the low byte wraps around to
$FF.



        ; 16-bit increment Word
        inc Word



        bne noinc inc Word+1
noinc:



        ; 16-bit decrement Word
        lda Word



        bne nodec dec Word+1
nodec:  dec Word



;16-bit increment shows even more advantage when used to
control a loop, because the 16-bit increment conveniently
leaves the zero flag set at the end only if the entire 16-bit
value is zero.
```

```
;6502 8 bit PRNG
;http://retro.hansotten.nl/6502-sbc/lee-davison-web-site/some-
code-bits/#prng
; by Lee Davison
; returns pseudo random 8 bit number in A. Affects A, X and Y.
; (r_seed) is the byte from which the number is generated and
MUST be
; initialised to a non zero value or this function will always
return
; zero. Also r_seed must be in RAM, you can see why......



rand_8



    LDA r_seed      ; get seed
    AND #$B8        ; mask non feedback bits
                ; for maximal length run with 8 bits we need
                ; taps at b7, b5, b4 and b3
    LDX #$05        ; bit count (shift top 5 bits)
    LDY #$00        ; clear feedback count
F_loop



    ASL A       ; shift bit into carry
    BCC bit_clr     ; branch if bit = 0



    INY         ; increment feedback count (b0 is XOR all the
                ; shifted bits from A)
bit_clr



    DEX         ; decrement count
    BNE F_loop      ; loop if not all done no_clr
```

```
TYA          ; copy feedback count
LSR A        ; bit 0 into Cb
LDA r_seed      ; get seed back
ROL A        ; rotate carry into byte
STA r_seed      ; save number as next seed
RTS          ; done r_seed


.byte   1       ; prng seed byte (must not be zero)
```

```
;6502 flag manipulation
;======================
;The Carry flag can be set cleared with SEC and CLC, but the
other flags have
;no instructions to do so.



;oVerflow flag
;-------------
;The 6502 has a CLV instruction to clear the oVerflow flag, but
has no
;complementary SEV instruction to set the V flag. However, with
careful
;arranging of code, you can do this by testing a byte of the
code.
;
;For example:



  BIT setv   ; V set from b6 of RTS opcode (also clears M)
  .setv



  RTS        ; opcode is %01100000



  BIT setv   ; V set from b6 of JMP opcode (also clears M)
  .setv



  JMP dest   ; opcode is %01001100



  BIT setv   ; V set from b6 of JMP opcode (also clears M)
  .setv



  JMP (vect) ; opcode is %01101100
```

```
  BIT setv   ; V set from b6 of NOP opcode (also sets M)
  .setv


  NOP        ; opcode is %11001010


  .bit
  LDA &FE08  ; where absaddr>=&C000
  BIT bit+2  ; V set from b14 of &FE08 (also sets M) .bit
  LDA &FD    ; where zpaddr>=&C0
  BIT bit+1  ; V set from b6 of &FD (also sets M)


;Minus flag
;----------
  .bit
  LDA &FE08  ; where absaddr>=&8000
  BIT bit+2  ; sets M from bit 15 &FE08


  .bit
  LDA &FD    ; where zpaddr>=&80
  BIT bit+1  ; sets M from bit 7 of &FD


  ORA #&80   ; sets M, changing A LDr #num   ; if num>&7F, sets
M changing register .clrm


  BIT clrm   ; opcode is %00101100, clears M


  AND #&7F   ; clears M, changing A LDr #num   ; if num<&80,
clears M changing register


;Zero flag
;---------
  CMP #num   ; if num known to be same as A, will set Z
```

```
  CMP #num   ; if num known to be different from A, will clear
Z
  LDr #0     ; sets Z, also clearing register


;Complement Carry
;----------------
  PHP
  PLA
  EOR #&01
  PHA
  PLP



  ROL A EOR #&01
  ROR A   ; also modifies M and Z



  ROR A EOR #&80
  ROL A   ; also modifies M and Z
```

```
;8-bit unsigned multiplication
;https://forums.atariage.com/topic/71120-6502-killer-hacks/page/2/#comment-896028



;Computing 8x8->16 multiply is more useful than 8x8->8, and isn't really any harder. The 6502's lack of add-without-carry is somewhat irksome, but there are a variety of workarounds that could be used.



; Compute mul1*mul2+acc -> acc:mul1 [mul2 is unchanged]



 ldx #8
 dec mul2



lp:
 lsr
 ror mul1



 bcc nope



 adc mul2



nope:
 dex
 bne lp



 inc mul2
```

```
;As for division, the normal approach is to do a shift-and-subtract. A 16/8->8+8
result is pretty easy, with the caveat that the results will be meaningless if the
quotient doesn't fit in eight bits. I'll try to work one up for you.
```

```
;8-bit number output in 6502 machine code
;========================================
;
;Print 8-bit hexadecimal
;======================
;Print value in A in hexadecimal padded with zeros to two
characters.
;
;    \ On entry  A=value to print
;    \ On exit   A corrupted
;    \ Size      22 bytes



    .PrHex


    PHA                         ;\ Save A
    LSR A:LSR A:LSR A:LSR A     ;\ Move top nybble to bottom
nybble
    JSR PrNybble                ;\ Print this nybble
    PLA                         ;\ Get A back and print bottom
nybble
    .PrNybble


    AND #15                     ;\ Keep bottom four bits
    CMP #10:BCC PrDigit         ;\ If 0-9, jump to print
    ADC #6                      ;\ Convert ':' to 'A'
    .PrDigit


    ADC #ASC"0":JMP OSWRCH      ;\ Convert to character and
print
```

```
;Print 8-bit hexadecimal (more cunning)
;=====================================
;Print value in A in hexadecimal padded with zeros to two
characters.
;(With reference to
http://www.obelisk.me.uk/6502/algorithms.html)
;
;    \ On entry  A=value to print
;    \ On exit   A corrupted
;    \ Size      21 bytes



    .PrHex



    PHA                         ;\ Save A
    LSR A:LSR A:LSR A:LSR A     ;\ Move top nybble to bottom
nybble
    JSR PrNybble



    PLA
    AND #15                     ;\ Mask out original bottom
nybble
    .PrNybble



    SED                         ;\ Switch to decimal arithmetic
    CLC
    ADC #&90                    ;\ Produce &90-&99+CC or &00-
&05+CS
    ADC #&40                    ;\ Produce &30-&39    or &41-&46
    CLD                         ;\ Switch back to binary
arithmetic
    JMP OSWRCH                  ;\ Print it
```

```
;Print 8-bit decimal 0-255
;========================
;Print value in A in decimal padded with zeros to three
characters.
;
;    \ On entry  A=value to print 0-255
;    \ On exit   A,X corrupted
;    \ Size      35 bytes



    .PrDec



    LDX #ASC"0"-1:SEC          ;\ Prepare for subtraction
    .PrDec100



    INX:SBC #100:BCS PrDec100  ;\ Count how many 100s
    ADC #100:JSR PrDecDigit    ;\ Print the 100s
    LDX #ASC"0"-1:SEC          ;\ Prepare for subtraction
    .PrDec10



    INX:SBC #10:BCS PrDec10    ;\ Count how many 10s
    ADC #10:JSR PrDecDigit     ;\ Print the 10s
    ORA #ASC"0":TAX            ;\ Pass 1s into X
    .PrDecDigit



    PHA:TXA:JSR OSWRCH         ;\ Print digit
    PLA:RTS                    ;\ Restore A and return







;Print 8-bit decimal 0-99
;========================
;Print value in A in decimal padded with zeros to two
```

```
characters.
;Works by converting into a BCD number and printing in hex.
;
;    \ On entry  A=value to print 0-99
;    \ On exit   A,X corrupted
;    \ Size      11 bytes + size of PrHex



    .PrDec


    TAX:LDA #&99                ;\ Move value to X, start at -1
in BCD
    SED                         ;\ Switch to decimal arithmetic
    .PrDecLp


    CLC:ADC #1                  ;\ Add one in BCD mode
    DEX:BPL PrDecLp             ;\ Loop for all of source number
    CLD                         ;\ Switch back to binary
arithmetic
    :
    ;\ Fall through into PrHex
   .PrHex
```

```
;8bit * 8bit = 16bit multiply
;https://codebase64.org/doku.php?id=base:8bit_multiplication_16bit_product



;Extended from https://codebase64.org/doku.php?
id=base:8bit_multiplication_8bit_product



;------------------------
; 8bit * 8bit = 16bit multiply
; By White Flame
; Multiplies "num1" by "num2" and stores result in .A (low byte, also in .X) and .Y
(high byte)
; uses extra zp var "num1Hi"



; .X and .Y get clobbered.  Change the tax/txa and tay/tya to stack or zp storage
if this is an issue.
;  idea to store 16-bit accumulator in .X and .Y instead of zp from bogax



; In this version, both inputs must be unsigned
; Remove the noted line to turn this into a 16bit(either) * 8bit(unsigned) = 16bit
multiply.



 lda #$00
 tay
 sty num1Hi  ; remove this line for 16*8=16bit multiply
 beq enterLoop







doAdd:
 clc
 adc num1
```

```
tax


 tya
 adc num1Hi


 tay
 txa


loop:
 asl num1


 rol num1Hi


enterLoop:  ; accumulating multiply entry point (enter with .A=lo, .Y=hi)
 lsr num2


 bcs doAdd


 bne loop


; 26 bytes
```

```
;
; Ullrich von Bassewitz, 2010-11-02
;
; CC65 runtime: 8x8 => 16 unsigned multiplication
;


        .export         umul8x8r16, umul8x8r16m


        .importzp       ptr1, ptr3









;----------------------------------------------------------------------------
; 8x8 => 16 unsigned multiplication routine.
;
;   LHS           RHS         result      result in also
; ------------------------------------------------------------
;   .A (ptr3-low)  ptr1-low     .XA           ptr1
;



umul8x8r16:
        sta     ptr3



umul8x8r16m:
        lda     #0              ; Clear byte 1
        ldy     #8              ; Number of bits
        lsr     ptr1            ; Get first bit of RHS into carry
@L0:    bcc     @L1
        clc
        adc     ptr3
```

```
@L1:   ror
       ror    ptr1



       dey
       bne    @L0
       tax
       stx    ptr1+1        ; Result in .XA and ptr1
       lda    ptr1          ; Load the result
       rts                  ; Done
```

```
;\ Calculating the day of the week for a given date
;\ ================================================
;\ Based on code at http://6502.org/source/misc/dow.htm by Paul
Guertin.
;\
;\ This routine works for any date from 1900-03-01 to 2155-12-
31.
;\ No range checking is done, so validate input before calling.
;\
;\ It uses the formula
;\     Weekday = (day + offset[month] + year + year/4 + fudge)
mod 7
;\       offset[month] adjusts the day count so 1st of a month
is effectively
;\         the (lastday+1)-th of the previous month.
;\       fudge is -1 when after 2099 as 2100 is not a leap
year.
;\
;\ On entry  A=day, 1..31
;\           X=month, 1..12
;\           Y=year-1900, 0..255
;\ On exit   A=day of week 0..6 for Sun..Sat, Carry will be Set
;\           Needs incrementing with ADC #0 after calling to
;\           become standard 1..7 range
;\ Size      45 bytes + 1 byte workspace
;\
.DayOfWeek



CPX #3:BCS dow_march    ;\ Year starts in March to bypass leap
year problem
DEY                     ;\ If Jan or Feb, decrement year
.dow_march



EOR #&7F                ;\ Invert A so carry works right
CPY #200                ;\ Carry will be 1 if 22nd century
ADC dow_months-1,X      ;\ A=day+month_offset
STA dow_tmp
```

```
TYA:JSR dow_mod7        ;\ Get the year MOD 7 to prevent
overflow
SBC dow_tmp:STA dow_tmp ;\ A=day+month_offset+year
TYA:LSR A:LSR A         ;\ Get the year DIV 4
CLC:ADC dow_tmp         ;\ A=day+month_offset+year+year/4, fall
through to MOD 7
.dow_mod7



ADC #7:BCC dow_mod7     ;\ Reduce A to A MOD 7
RTS
.dow_months



EQUB 1:EQUB 5:EQUB 6:EQUB 3  ;\ Month offsets
EQUB 1:EQUB 5:EQUB 3:EQUB 0
EQUB 4:EQUB 2:EQUB 6:EQUB 4
.dow_tmp



EQUB 0                  ;\ Temporary storage
;\
;\
;\ You can test this with:
;\
;\ FOR Y%=1 TO 255
;\ FOR X%=1 TO 12
;\ FOR A%=1 TO 31
;\   PRINT A%;"/";X%;"/";1900+Y%;" ";
;\   PRINT MID$("SunMonTueWedThuFriSat",((USRDayOfWeek) AND
&FF)*3+1,3)
;\ NEXT A%:NEXT X%:NEXT Y%
;\
;\ I must say that this is an impressive bit of code!
;\
```

```
;Counting, incrementing, decrementing
;====================================
;This will count from any value and end at zero: CMP #1
;\ Carry set if A>0
    ADC #0          ;\ If A>0, A=A+1; if A=0, A=A+0



;This code will count down from any value and ends at &FF: CMP
#&FF        ;\ Carry set if A=&FF
    SBC #0          ;\ If A<&FF, A=A-1; if A=&FF, A=A-0







;Comparing various values
;========================
    ORA #0          ;\ Sets EQ if A=0, the usual test for zero
CMP #1          ;\ Sets CC if A=0, sets CS if A<>0
```

```
;Decrementing 16-bit and larger numbers
;=====================================
;Incrementing a 16-bit or larger number in memory is trivial,
as the Zero
;flag is set when a byte wraps from &FF to &00:


    INC addr+0
    BNE done


    INC addr+1
    BNE done


    INC addr+2
    BNE done


    INC addr+3
    .done




;However, decrementing is not as simple, as you need to check
for the wrap
;from &00 to &FF. The following does not work:


    DEC num+0
    BNE done


    DEC num+1
    BNE done
```

```
    DEC num+2
    BNE done


    DEC num+3
    .done




;Each byte being decremented must be checked /before/ it is
decremented. This
;requires a register to be corrupted.


    LDA num+0:BNE skip    ;\ Skip if low byte not about to wrap
from &00 to &FF
    DEC num+1             ;\ Decrement high byte
    .skip


    DEC num+0             ;\ Decrement low byte



;    \ The following code tests for zero
    ORA num+1
    BNE loop




;This can be expanded for 24-bit, 32-bit, and larger counters:


    LDA num+0:BNE dec0    ;\ Skip past if b0-b7 not about to
wrap
```

```
    LDA num+1:BNE dec1    ;\ Skip past if b8-b15 not about to
wrap
    LDA num+2:BNE dec2    ;\ Skip past if b16-b24 not about to
wrap
    DEC num+3             ;\ Decrement b24-b31
    .dec2



    DEC num+2             ;\ Decrement b16-23
    .dec1



    DEC num+1             ;\ Decrement b8-b15
    .dec0



    DEC num+0             ;\ Decrement b0-b7



;    \ Test if num=0
    ORA num+1
    ORA num+2
    ORA num+3
    BNE loop
```

```
;The temporary register can be A, X or Y, whichever is most
convenient.



;You can use the same method to do a 16-bit decrement of a
value in the XY registers:



    TXA:BNE skip          ;\ Skip if low byte not about to wrap
from &00 to &FF
```

```
    DEY                 ;\ Decrement high byte
    .skip


    DEX                 ;\ Decrement low byte



;    \ The following code tests for zero
    BNE loop


    TYA
    BNE loop




;This is the complement to a 16-increment in XY:


    INX:BNE done


    INY
    .done




;or
    INX
        BNE loop
```

```
    INY
    BNE loop




;With reference to
http://www.obelisk.me.uk/6502/algorithms.html.
```

;http://6502.org/source/general/SWN.html
;David Galloway made this suggestion on the facebook 6502 Programming group, for swapping nybbles. $36 becomes $63, $A1 becomes $1A, etc.. It takes only 8 bytes and 12 clock cycles, and no variables, no stack usage, no look-up table, no X or Y usage. It uses only the accumulator and status register.

```
        ASL  A ADC  #$80
        ROL  A


        ASL  A ADC  #$80
        ROL  A
```

;Straight-lining it takes only five bytes more than a subroutine call and cuts the execution time in half. It could of course be put in a macro. How that is done exactly will depend on your assembler, but might go something like: SWN: MACRO

```
        ASL  A ADC  #$80
        ROL  A


        ASL  A ADC  #$80
        ROL  A
```

ENDM

```
 ;-------------
;and would be called simply with SWN as if it were an assembly-
language mnemonic. You probably won't use it many times in a
program anyway for the straight-lining to take up appreciable
memory, but you might want it pretty fast when you do.
```

```
;Efficient sign extension
;http://forum.6502.org/viewtopic.php?f=2&t=6069



;I was wandering the 'net and somehow kept noticing that more
than a few 65xx programmers were taking more effort than
necessary to sign-extend integers. I've been playing with 65xx
assembly for at least 39 years, and I really can't remember if
I got this little ditty from somewhere else long ago and simply
forgot from where, or if it's an original idea of mine. Either
way, here it is, all ten bytes and 13 cycles worth:



; Sign-extend A to 16-bits in ZP
    sta  ZP



    lda  #$7f
    cmp  ZP   ; cc for <0, cs for >=0
    sbc  #$7f
    sta  ZP+1 ; $ff for <0, $00 for >=0



Code:
; Promote int16 in ZP to int32
    lda  #$7f
    cmp  ZP+1 ; cc for <0, cs for >=0
    sbc  #$7f
    sta  ZP+2 ; $ff for <0, $00 for >=0
    sta  ZP+3



;----------------------------------------



;With CMOS instructions, I can cut the 8->16 case down to 9
bytes with the same average cycle count (either 11 or 15
cycles, depending on sign):
  STA zp
```

```
  STZ zp+1
  ASL A ; carry set for negative
  BCC :+
  DEC zp+1
: ; continue


;You could avoid the ASL A if the flags already reflect the
contents of A, and branch on BPL instead of BCC. This saves 1
more byte and 2 cycles. This even has the virtue of preserving
the value in A.
;For the 16->32 case:


  LDA #0
  BIT zp+1 ; N flag reflects value in memory
  BPL :+
  DEC A


: STA zp+2
  STA zp+3


;This is 11 bytes, best-case 14 cycles, worst-case 15 cycles.
Not quite so good - but if we delete the final STA (thus
extending the sign by only one byte), we are back to 9 bytes
and 11-12 cycles.


;----------------------------------------


;The Sign-extend section the Nesdev Wiki's synthetic
instructions page offers a similar technique for a constant-
time replacement of A with its sign extension, but this one
uses no branches. If one bookends it with a STA zp and a STA
zp+1, it's almost (but not quite, by 1 byte and 1 cycle) as
good as yours. It might be useful if you don't want to store A
```

in the zero page, though, since it doesn't rely on re-using the original value of A as you need to do with your CMP.

```
      ; Constant-time version, but destroys the carry
            ASL A           ; sign bit into carry; use CPX etc.
if using X reg
            LDA #$00
            ADC #$FF        ; C set:   A = $FF + C = $00
                            ; C clear: A = $FF + C = $FF
            EOR #$FF        ; Flip all bits and they all now
match C
```

```
;http://6502.org/source/integers/fastx10.htm
;Fast Multiply by 10
;By Leo Nechaev (leo@ogpi.orsk.ru), 28 October 2000.
MULT10  ASL         ;multiply by 2
        STA TEMP    ;temp store in TEMP
        ASL         ;again multiply by 2 (*4)
        ASL         ;again multiply by 2 (*8)
        CLC
        ADC TEMP    ;as result, A = x*8 + x*2
        RTS



TEMP    .byte 0
;Last page update: November 6, 2001.
```

```
;General 8bit * 8bit = 8bit multiply
;https://codebase64.org/doku.php?id=base:8bit_multiplication_8bit_product ;
General 8bit * 8bit = 8bit multiply
; by White Flame 20030207



; Multiplies "num1" by "num2" and returns result in .A
; Instead of using a bit counter, this routine early-exits when num2 reaches zero,
thus saving iterations.







; Input variables:
;   num1 (multiplicand)
;   num2 (multiplier), should be small for speed
;   Signedness should not matter ; .X and .Y are preserved
; num1 and num2 get clobbered lda #$00
 beq enterLoop



doAdd:
 clc
 adc num1







loop:
 asl num1



enterLoop: ;For an accumulating multiply (.A = .A + num1*num2), set up num1
and num2, then enter here
 lsr num2



 bcs doAdd bne loop
```

```
end:
; 15 bytes
```

```
;INC and DEC techniques
;http://6502org.wikidot.com/software-incdec
;Typical 16-bit increment
;Overflow (incrementing to $0000) sets the Z flag is set (i.e.
BEQ branches).



;
      INC NUML


      BNE LABEL


      INC NUMH
```

LABEL

```
;Typical 16-bit decrement
;
      LDA NUML


      BNE LABEL


      DEC NUMH


LABEL DEC NUML
```

```
;16-bit decrement, test for zero
;
      LDA NUML


      BNE LABEL


      LDA NUMH


      BEQ ZERO ; branch when NUM = $0000 (NUM is not
decremented in that case)
      DEC NUMH
```

```
LABEL DEC NUML




;Add 255
;
      LDA NUML


      BEQ LABEL


      INC NUMH


LABEL DEC NUML




;Subtract 255
;
      INC NUML


      BEQ LABEL


      DEC NUMH
```

LABEL

```
;Constant time increment 6 cycles
;A = high byte, X = low byte
;Overflow (incrementing to $0000) sets the carry


CPX #$FF
INX
ADC #$00


;Constant time decrement 6 cycles
;A = high byte, X = low byte
;Underflow (decrementing to $FFFF) clears the carry


CPX #$01
DEX
SBC #$00


;Increment and stop at $00
;This counts up to $FF then to $00, then stays at $00.


CMP #1
ADC #0


;One advantage of a CMP before the ADC (and SBC below) is that
the accumulator always contains the correct value. If it were
an increment, compare, branch, and decrement (to return to the
```

stop value), then it would briefly, but temporarily, not be at the stop value.

```
;Decrement and stop at $FF
;This counts down to $00 then to $FF, then stays at $FF.

CMP #$FF
SBC #0

;Double-decrement
;To subtract two from a 16-bit value, you can execute an above routine twice, but it is probably more efficient to code it as its own case: DEC2  LDA NUML

      SEC
      SBC #2
      STA NUML

      BCS LABEL

      DEC NUMH
```

LABEL

;If for any reason you may need to preserve the accumulator or V flag, or if you need a binary decrement while D is set, you can wrap the above routine with PHP PHA CLD … and … PLA PLP, or you can sacrifice the X or Y register like so: DEC2 LDY NUML

```
      CPY #2
      DEY
      DEY
      STY NUML


      BCS LABEL


      DEC NUMH
```

LABEL

```
;If you know you're not in decimal mode, you can keep the high
byte in A and the low byte in X or Y, resulting in a tidy: DEC2
CPY #2
      DEY
      DEY
      SBC #0
```

```
;Integer Square Roots in 6502 machine code
;=========================================
;A simple, but inefficient way of calculating integer square
roots is to
;count how many times increasing odd numbers can be subtracted
from the
;starting value. For example:
;
;32-1=31 -> 31-3=28 -> 28-5=23 -> 23-7=16 -> 16-9=7 -> 7-11<0
;        1          2          3          4          5
;
;-> SQR(30)=5, remainder 7
;
;This can be done in 6502 machine code as follows:
;
;\ Calculate 16-bit square root
;\ ----------------------------
;\ On entry  num+0..num+1   = input value
;\           sub+0..sub+1 = workspace
;\ On exit   X = SQR(input value)
;\           Y = remainder from SQR(input value)
;\           A,num,sub = corrupted
;\ Size      37 bytes
;\
.sqr                                ;\ On entry, !num=input value
LDX #1:STX sub+0:DEX:STX sub+1      ;\ Initialise sub to first
subtrand
                                    ;\ and initialise X to SQR(0)
.sqr_loop                           ;\ Repeatedly subtract
increasing
SEC                                 ;\   odd numbers until num<0
LDA num+0:TAY:SBC sub+0:STA num+0 ;\ num=num-subtrand,
remainder in Y
LDA num+1:SBC sub+1:STA num+1
BCC sqr_done                        ;\ num<0, all done
INX                                 ;\
LDA sub+0:ADC #1:STA sub+0          ;\ step +2 to next odd number
BCC sqr_loop                        ;\ no overflow, subtract
again
INC sub+1:BNE sqr_loop              ;\ INC high byte and subtract
again
.sqr_done
```

```
RTS                                 ;\ X=root, Y=remainder
:


;You can test it with:
;  FOR A%=1 to 65535:!num=A%:!sub=USR sqr:PRINT A%,sub?1,sub?
2:NEXT
;
;or with:
;  FOR A%=1 to 65535
;    !num=A%:B%=USR sqr
;    PRINT A%,(B%AND&FF00)DIV256,(B%AND&FF0000)DIV65536
;  NEXT
;
;It is simple to reduce the code to calculate roots of 8-bit
numbers by
;removing the code to subtract in+1. The code uses an 8-bit
counter for the
;root, so to calculate roots of numbers larger than 16 bits
different code is
;needed, as the square root of &10000 (a 17-bit number) is &100
(a 9-bit
;number).
```

```
;Jump indirect using stack ;An indirect address can be pushed
on the stack and then jumped to by
;returning. Since the address is on the stack, no temporary
locations have to
;be assigned for the destination address and the code is re-
entrant. Normal
;return increments the address, but rti doesn't: lda    #$12
; push high byte first
        pha
        lda     #$34
        pha
        php
        rti             ; jumps to $1234
```

```
; Linear congruential pseudo-random number generator
;http://6502.org/source/integers/random/random.html
;
; Calculate SEED = 1664525 * SEED + 1
;
; Enter with:
;
;   SEED0 = byte 0 of seed
;   SEED1 = byte 1 of seed
;   SEED2 = byte 2 of seed
;   SEED3 = byte 3 of seed
;
; Returns:
;
;   SEED0 = byte 0 of seed
;   SEED1 = byte 1 of seed
;   SEED2 = byte 2 of seed
;   SEED3 = byte 3 of seed
;
; TMP is overwritten
;
; For maximum speed, locate each table on a page boundary
;
; Assuming that (a) SEED0 to SEED3 and TMP are located on page
zero, and (b)
; all four tables start on a page boundary:
;
;   Space: 58 bytes for the routine
;          1024 bytes for the tables
;   Speed: JSR RAND takes 94 cycles
;
RAND     CLC       ; compute lower 32 bits of:
         LDX SEED0 ; 1664525 * ($100 * SEED1 + SEED0) + 1
         LDY SEED1



         LDA T0,X
         ADC #1
         STA SEED0



         LDA T1,X
         ADC T0,Y
```

```
          STA SEED1



          LDA T2,X
          ADC T1,Y
          STA TMP



          LDA T3,X
          ADC T2,Y
          TAY       ; keep byte 3 in Y for now (for speed)
          CLC       ; add lower 32 bits of:
          LDX SEED2 ; 1664525 * ($10000 * SEED2)
          LDA TMP



          ADC T0,X
          STA SEED2



          TYA
          ADC T1,X
          CLC
          LDX SEED3 ; add lower 32 bits of:
          ADC T0,X  ; 1664525 * ($1000000 * SEED3)
          STA SEED3



          RTS
;
; Generate T0, T1, T2 and T3 tables
;
; A different multiplier can be used by simply replacing the
four bytes
; that are commented below
;
; To speed this routine up (which will make the routine one
byte longer):
; 1. Delete the first INX instruction
; 2. Replace LDA Tn-1,X with LDA Tn,X (n = 0 to 3)
; 3. Replace STA Tn,X with STA Tn+1,X (n = 0 to 3)
; 4. Insert CPX #$FF between the INX and BNE GT1
```

```
;
GENTBLS  LDX #0      ; 1664525 * 0 = 0
         STX T0


         STX T1


         STX T2


         STX T3


         INX
         CLC
GT1      LDA T0-1,X  ; add 1664525 to previous entry to get
next entry
         ADC #$0D    ; byte 0 of multiplier
         STA T0,X
         LDA T1-1,X
         ADC #$66    ; byte 1 of multiplier
         STA T1,X
         LDA T2-1,X
         ADC #$19    ; byte 2 of multiplier
         STA T2,X
         LDA T3-1,X
         ADC #$00    ; byte 3 of multiplier
         STA T3,X
         INX         ; note: carry will be clear here
         BNE GT1
```

RTS

```
;The short version is just an ordinary 32-bit * 32-bit
multiplication routine. The DB pseudo-op is called .BYTE on
some assemblers. Consult the assembler documentation for the
pseudo-op name it expects.


; Linear congruential pseudo-random number generator
;
; Calculate SEED = 1664525 * SEED + 1
;
; Enter with:
;
;   SEED0 = byte 0 of seed
;   SEED1 = byte 1 of seed
;   SEED2 = byte 2 of seed
;   SEED3 = byte 3 of seed
;
; Returns:
;
;   SEED0 = byte 0 of seed
;   SEED1 = byte 1 of seed
;   SEED2 = byte 2 of seed
;   SEED3 = byte 3 of seed
;
; TMP, TMP+1, TMP+2 and TMP+3 are overwritten
;
; Note that TMP to TMP+3 and RAND6 are high byte first, low
byte last
```

```
;
; Assuming that (a) SEED0 to SEED3 and TMP+0 to TMP+3 are all
located on page
; zero, and (b) none of the branches cross a page boundary:
;
;   Space: 53 bytes
;   Speed: JSR RAND takes 2744 cycles, on average (1624 to 3864
cycles)
;          specifically, JSR RAND takes 1624 + 70 * N cycles,
where
;          N = number of bits of SEED that are 1
;
RAND     LDA #1              ; store 1 in TMP
         LDX #3
RAND1    STA TMP,X
         LSR
         DEX
         BPL RAND1



         LDY #$20            ; calculate SEED = SEED * RAND4 +
TMP
         BNE RAND5           ; branch always
RAND2    BCC RAND4           ; branch if a zero was shifted out
         CLC                 ; add multiplier to product
         LDX #3
RAND3    LDA TMP,X
         ADC RAND4,X
         STA TMP,X
         DEX
         BPL RAND3



RAND4    ROR TMP             ; shift result right
         ROR TMP+1
         ROR TMP+2
         ROR TMP+3
RAND5    ROR SEED3           ; shift out old seed, and shift in
new seed
         ROR SEED2



         ROR SEED1
```

```
         ROR SEED0


         DEY
         BPL RAND2


         RTS
RAND6    DB  $00,$19,$66,$0D ; multiplier (high byte first!)
```

;Here is the 16-bit version of RANDOM, called RANDOM16.

```
; Linear congruential pseudo-random number generator
;
; Get the next SEED and obtain an 16-bit random number from it
;
; Requires the RAND subroutine
;
; Enter with:
;
;   MOD = modulus
;
; Exit with:
;
;   RND = random number, 0 <= RND < MOD
;
; TMP is overwritten, but only after RND is called.
;
RANDOM16 JSR RAND  ; get next seed
         LDA #0    ; multiply SEED by MOD
         STA RND+1
         STA RND
```

```
          STA TMP


          LDY #16
R16A      LSR MOD+1 ; shift out modulus
          ROR MOD


          BCC R16B  ; branch if a zero was shifted out
          CLC       ; add SEED, keep upper 16 bits of product in
RND
          ADC SEED0


          TAX
          LDA TMP


          ADC SEED1


          STA TMP


          LDA RND


          ADC SEED2


          STA RND


          LDA RND+1
          ADC SEED3
```

```
          STA RND+1
          TXA
R16B      ROR RND+1 ; shift product right
          ROR RND



          ROR TMP



          ROR
          DEY
          BNE R16A



          RTS
```

```
;minimum code pseudo sine wave
;http://forum.6502.org/viewtopic.php?f=2&t=2404



;This is meant to be the standard parabolic pseudo sine wave
generator. In the absolute minimum of code.
 clc
 ldy #$10
 lda #$7F
LOOP1



 dey
LOOP2



 sty temp adc temp bmi LOOP1



 iny
 jmp LOOP2









;I guess if you were going to time it with code you'd have to
add a nop to the dey side.



 clc
 ldy #$10
 lda #$7F
LOOP1
```

```
 nop
 dey
LOOP2


 sty temp adc temp bmi LOOP1


 iny
 jmp LOOP2
```

```
;Permutation Generator by Paul Guertin
;http://6502.org/source/integers/perm.htm


;How to generate permutations in 6502 assembly.
;By Paul Guertin (pg@sff.net), 19 August 2000.
;If you have n distinct elements, there are n! ways of
arranging them in order. For example, the 3!=6 permutations of
the digits "123" are 123, 213, 312, 132, 231, and 321.


;Generating permutations is usually done with a recursive
procedure, but here is a cute iterative routine that does it
simply and efficiently. One caveat: permutations are not
generated in lexicographical order, but in an order such that
two successive permutations differ by exactly one swap (as in
the list above).


;To keep this routine as generic as possible, it calls two
user-supplied subroutines: EXCHANGE, which swaps elements X and
Y, and PROCESS, which does something with the permutation (such
as print it). This way, you can easily permute any data set.


SIZE     EQU 4           ; Number of elements to permute
TEMP     EQU $6          ; (1 byte) Temporary storage


PERMGEN:
         LDA #0          ; Clear the stack
         LDX #SIZE-1
CLRSTK   STA STK,X
         DEX
         BPL CLRSTK


         BMI START       ; Do first permutation
```

```
LOOP     LDA STK,X
         STX TEMP



         CMP TEMP        ; Swap two elements if stk,x < x
         BCS NOEXCH      ; else just increment x
         INC STK,X
         TAY
         TXA
         LSR             ; Check whether x is even or odd
         BCS XODD        ;  x odd  -> swap x and stk,x
         LDY #0          ;  x even -> swap x and 0
XODD     JSR EXCHANGE    ; Swap elements x and y (user-
supplied)
START    JSR PROCESS     ; Use the permutation (user-supplied)
         LDX #1          ;
         BNE LOOP        ; (always)



NOEXCH   LDA #0          ; No exchange this pass,
         STA STK,X       ; so we go up the stack
         INX
         CPX #SIZE



         BCC LOOP        ; Loop until all permutations
generated
         RTS
STK      DS  SIZE        ; Stack space (ds reserves "size"
bytes)
;Example of use:
;print all permutations of integers {1, 2, ..., SIZE}
EXAMPLE  LDX #SIZE-1     ; Set up ASCII digit string
         CLC
EINIT    TXA
         ADC #"1"
         STA DIGIT,X
         DEX
         BPL EINIT
```

```
         JMP PERMGEN     ; Jump to permutation generator
DIGIT    DS  SIZE



;Here are the two sub-routines:
EXCHANGE LDA DIGIT,X     ; Swap two digits in the string
         PHA
         LDA DIGIT,Y
         STA DIGIT,X
         PLA
         STA DIGIT,Y
         RTS



PROCESS  LDX #0          ; Print the digit string (Apple II
specific)
PLOOP    LDA DIGIT,X
         JSR $FDED       ; Print accumulator as ASCII character
         INX
         CPX #SIZE



         BCC PLOOP



         JMP $FD8E       ; Print a carriage return
;Last page update: August 19, 1999.
```

```
;Practical Memory Move Routines by Bruce Clark
;http://6502.org/source/general/memory_move.html
;
;Here are some reasonably fast general-purpose routines for
moving blocks of memory. You simply specify the address to move
from, the address to move to, and the size of the block. When
SIZE is zero, no bytes are moved. SIZEL and SIZEH do not need
to be consecutive memory locations, or even on the zero page
for that matter. These routines only take one additional cycle
if SIZEL is not on the zero page. Likewise, they only take one
additional cycle if SIZEH is not on the zero page. Note that
this adds only to the total number of cycles, not to the number
of cycles per byte, since neither SIZEL nor SIZEH is inside a
loop anywhere.
;
;These routines are intended to be both flexible and practical,
without being excessively lengthy or excessively slow. To that
end, they can be placed in ROM or in RAM.
;
;There are three routines moving memory upward (i.e. to a
higher address), each of which is tailored to a slightly
different set of input parameters.
;
; Move memory down
;
; FROM = source start address
;   TO = destination start address
; SIZE = number of bytes to move
;
MOVEDOWN LDY #0
         LDX SIZEH



         BEQ MD2



MD1      LDA (FROM),Y ; move a page at a time
         STA (TO),Y
         INY
         BNE MD1
```

```
         INC FROM+1
         INC TO+1
         DEX
         BNE MD1


MD2      LDX SIZEL


         BEQ MD4


MD3      LDA (FROM),Y ; move the remaining bytes
         STA (TO),Y
         INY
         DEX
         BNE MD3


MD4      RTS


; Move memory up
;
; FROM = source start address
;   TO = destination start address
; SIZE = number of bytes to move
;
MOVEUP   LDX SIZEH    ; the last byte must be moved first
         CLC          ; start at the final pages of FROM and TO
         TXA
         ADC FROM+1
         STA FROM+1
         CLC
         TXA
         ADC TO+1
         STA TO+1
         INX          ; allows the use of BNE after the DEX
below
         LDY SIZEL
```

```
          BEQ MU3


          DEY          ; move bytes on the last page first
          BEQ MU2


MU1       LDA (FROM),Y
          STA (TO),Y
          DEY
          BNE MU1


MU2       LDA (FROM),Y ; handle Y = 0 separately
          STA (TO),Y
MU3       DEY
          DEC FROM+1   ; move the next page (if any)
          DEC TO+1
          DEX
          BNE MU1
```

RTS

```
; Move memory up
;
; FROM = 1 + source end address
; TO   = 1 + destination end address
; SIZE = number of bytes to move
;
MOVEUP   LDY #$FF
         LDX SIZEH



         BEQ MU3



MU1      DEC FROM+1
         DEC TO+1
MU2      LDA (FROM),Y ; move a page at a time
         STA (TO),Y
         DEY
         BNE MU2



         LDA (FROM),Y ; handle Y = 0 separately
         STA (TO),Y
         DEY
         DEX
         BNE MU1



MU3      LDX SIZEL



         BEQ MU5
```

```
         DEC FROM+1
         DEC TO+1
MU4      LDA (FROM),Y ; move the remaining bytes
         STA (TO),Y
         DEY
         DEX
         BNE MU4


MU5      RTS


; Move memory up
;
; FROM = source end address
; TO   = destination end address
; SIZE = number of bytes to move
;
MOVEUP   LDY #0
         LDX SIZEH


         BEQ MU3


MU1      LDA (FROM),Y ; handle Y = 0 separately
         STA (TO),Y
         DEY
         DEC FROM+1
         DEC TO+1
MU2      LDA (FROM),Y ; move a page at a time
         STA (TO),Y
         DEY
         BNE MU2


         DEX
         BNE MU1


MU3      LDX SIZEL
```

```
          BEQ MU5



          LDA (FROM),Y ; handle Y = 0 separately
          STA (TO),Y
          DEY
          DEX
          BEQ MU5



          DEC FROM+1
          DEC TO+1
MU4       LDA (FROM),Y ; move the remaining bytes
          STA (TO),Y
          DEY
          DEX
          BNE MU4



MU5       RTS
;Even more speed can be gained by using self-modifying code,
i.e. replacing the (ZeroPage),Y addressing mode with the
Absolute,Y addressing mode. This will take 2 fewer cycles per
byte. There will be some additional cycles from the added
instructions that self-modify the code, but the self-
modification occurs only once, and therefore adds these cycles
to total number of cycles, rather than the number of cycles per
byte moved. As always, the instructions that are self-modified
can't be located in ROM.
;Last page update: April 3, 2004.
```

```
; Reverse the byte stored at $02
; Answer returned in accumulator
revbyte  ldx #$07
loop1    asl $02
         ror
         dex
         bpl loop1
         rts
```

```
;Shifting and Rotating bits
;==========================
;The 6502's rotate instructions rotate nine bits through the carry flag. You
;can rotate eight bits with the following instructions: ;    \ RLC A - 8-bit rotate left circular, leaves Carry=old bit 7
                ;\ Cy A=abcdefgh
    CMP #&80    ;\  a   abcdefgh
    ROL A       ;\  a   bcdefgha ;    \ RLC A - 8-bit rotate left circular, leaves Carry clear
                ;\ Cy A=abcdefgh
    ASL A       ;\  a   abcdefg0
    ADC #0      ;\  0   bcdefgha ;    \ RRC A - 8-bit rotate right circular, leaves Carry=old bit 0
                ;\ Cy A=abcdefgh
    PHA         ;\  ?   abcdefgh
    ROR A       ;\  h   ?abcdefg
    PLA         ;\  h   abcdefgh
    ROR A       ;\  h   habcdefg ;The 6502's shift instructions are logical shifts with a zero bit entering
;the data to replace bits moved out. You can do an arthimetic shift with the
;following instructions:



;    \ ASR A - Arithmetic shift right - new bit 7 is the same as old bit 7
                ;\ Cy A=abcdefgh
    CMP #&80    ;\  a   abcdefgh
    ROR A       ;\  h   aabcdefg ;The following rotates two bits through an 8-bit left circular rotate: ;    \ Rotate two bits left through A
                ;\ Cy A=abcdefgh
    ASL A       ;\  a   bcdefgh0
    ADC #&80    ;\  b   Bcdefgha
    ROL A       ;\  B   cdefghab ;This gives an efficient way to swap the two nybbles in the A register: ;    \ SWP A - swap nybbles
                ;\ Cy A=abcdefgh
    ASL A       ;\  a   bcdefgh0
    ADC #&80    ;\  b   Bcdefgha
    ROL A       ;\  B   cdefghab
    ASL A       ;\  c   defghab0
```

```
    ADC #&80   ;\  d   Defghabc
    ROL A      ;\  D   efghabcd


;Setting, Clearing and Copying bits of data
;=========================================
;AND xx will clear the bits in A that are also clear in xx, for
example:
;       A            xx      after AND xx
;   %abcdefgh    %01010101    %0b0d0f0h ;ORA xx will set the
bits in A that are also set in xx, for example:
;       A            xx      after ORA xx
;   %abcdefgh    %01010101    %a1c1e1g1


;EOR xx will toggle the bits in A that are set in xx, for
example:
;       A            xx      after EOR xx
;   %abcdefgh    %01010101    %aBcDeFgH


;To clear the bits in A that are ''set'' in xx, use both ORA
and EOR, for example:
;       A            xx      after ORA xx  after EOR xx
;   %abcdefgh    %01010101    %a1c1e1g1    %a0c0e0g0


;You can copy a number of bits to a memory location without
changing the
;other bits using EOR and AND. For example, to copy the top
four bits of A
;into a memory location without changing the bottom four bits,
use the
;following:


;               A=12345678  dst=abcdefgh
    EOR dst;      ********      abcdefgh
    AND #&F0;     ****0000      abcdefgh
    EOR dst;      1234efgh      abcdefgh
    STA dst;      1234efgh      1234efgh ;This is much more
```

```
efficient than the usual code: PHA
    LDA dst


    AND #&0F


    STA dst


    PLA
    AND #&F0


    ORA dst


    STA dst
```

```
;Swapping data
;=============
;You can swap the contents of two memory locations by EORing
them with each other: ;    \ SWP addr1,addr2 - swap contents of
addr1 and addr2
    LDA addr1


    EOR addr2


    STA addr1


    EOR addr2
```

```
    STA addr2


    EOR addr1


    STA addr1




;This is in contrast to the usual way of using a temporary
variable: ;    \ Swapping two bytes of data with a temporary
memory location
    LDA addr1


    STA tmp


    LDA addr2


    STA addr1


    LDA tmp


    STA addr2
```

```
;    \ Swapping two bytes of data with a temporary register
    LDX addr1


    LDA addr2


    STA addr1


    STX addr2
```

```
;https://www.nesdev.org/wiki/Synthetic_instructions ;Arithmetic shift right
;The ARM instruction set has an arithmetic right shift, which doesn't alter the sign (top) bit. This shift is used to divide a signed value by two. But the 6502 lacks this instruction; LSR doesn't work because it shifts the sign bit to the right, then clears it.
;To implement this, we need carry set to the sign bit, then we can use ROR. CMP #$80 performs this task; if the value is less than $80, carry is cleared, otherwise it's set: ; Arithmetic shift right A
cmp #$80
ror a




;If the operand is in memory, we just use ASL to move the sign bit into carry: ; Arithmetic shift right Value
lda Value


asl a


ror Value









;8-bit rotate
;The 65xx series rotate instructions are all 9-bit, not 8-bit as often imagined. If they really were 8-bit, then eight ROR or ROL instructions in a row would leave A with its original
```

value. In actuality, nine are required to do so, since the carry acts as a ninth bit of A.
;Similar to arithmetic right shift, we must set carry to the top or bottom bit in advance of the rotate. For 8-bit rotate left, it's simple: ; 8-bit rotate left A
cmp #$80
rol a

; alternate method
asl a

adc #0

;For 8-bit rotate right, we must save and restore A: ; 8-bit rotate right A
pha
lsr a

pla
ror a

;A could be saved and restored using other methods, like TAX and TXA, etc.
;Branching can also be used:

```
    ; 8-bit rotate right A
    lsr a
```

```
    bcc skip

    adc #$80-1 ; carry is set, so will add extra 1
skip:

;If the operand is in memory:

; 8-bit rotate left Value
lda Value

asl a

rol Value

; 8-bit rotate right Value
lda Value

lsr a

ror Value
```

;Sign-extend
;https://www.nesdev.org/wiki/Synthetic_instructions ;When increasing the number of bits in a signed value, the new high bits are filled with copies of the sign bit. This is called sign extension. For example, sign-extending the 8-bit value $80 (-128) to 16 bits sets the new bits, resulting in $FF80; sign-extending $7F (+127) to 16 bits results in $007F. The following sequences calculate the upper byte of the sign-extended value.

```
        ; calculates upper byte of sign-extension of A
        ora #$7F
        bmi neg

        lda #0
neg:
        ; calculates upper byte of sign-extension of A, alternate version
        and #$80
        bpl pos

        lda #$ff
pos:
```

;This constant-time version (7 bytes, 8 cycles) destroys the carry, so don't try using it in the middle of a multi-byte addition: asl a   ; cpx #$80 or cpy #$80 is also possible

```
  lda #$00
  adc #$FF  ; C is unchanged and A = $00 if C was set or $FF if C was clear
  eor #$FF  ; now all bits of A are set to what bit 7 was ;If you're just trying to add an 8-bit delta to a 16-bit value, you could try subtracting 256 from the value by decrementing the high byte if the value is negative and then adding as if it were unsigned.

  lda delta_value bpl notneg
```

```
dec value_hi notneg:
clc
adc value_lo



sta value_lo lda #0
adc value_hi



sta value_hi
```

```
;Software - Math - Integer Division
;http://6502org.wikidot.com/software-math-intdiv



;8-bit / 8-bit = 8-bit quotient, 8-bit remainder (unsigned)
;Inputs:
;TQ = 8-bit numerator
;B = 8-bit denominator
;Outputs:
;TQ = 8-bit quotient of TQ / B
;accumulator = remainder of TQ / B
;
   LDA #0
   LDX #8
   ASL TQ



L1 ROL
   CMP B



   BCC L2



   SBC B



L2 ROL TQ



   DEX
   BNE L1









;16-bit / 8-bit = 8-bit quotient, 8-bit remainder (unsigned)
;Inputs:
;TH = bits 15 to 8 of the numerator
;TLQ = bits 7 to 0 of the numerator
```

```
;B = 8-bit denominator
;Outputs:
;TLQ = 8-bit quotient of T / B
;accumulator = remainder of T / B
;
   LDA TH



   LDX #8
   ASL TLQ


L1 ROL
   BCS L2


   CMP B


   BCC L3


L2 SBC B


;
; The SEC is needed when the BCS L2 branch above was taken
;
   SEC
L3 ROL TLQ


   DEX
   BNE L1
```

```
;Square Calculator
;http://retro.hansotten.nl/6502-sbc/lee-davison-web-site/some-code-bits/#prng
;This routine calculates the 16-bit unsigned integer square of a signed 16-bit integer in the range +/-255 (decimal).



; by Lee Davison; Calculates the 16 bit unsigned integer square of the signed
; 16 bit integer in Numberl/Numberh.
; The result is always in the range 0 to 65025 and is held in Squarel/Squareh
;
; The maximum input range is only +/-255 and no checking is done to ensure that
; this is so.
;
; This routine is useful if you are trying to draw circles as for any circle
;
; x^2+y^2=r^2 where x and y are the co-ordinates of any point on the circle and
; r is the circle radius
;
; Destroys all registers .ORG    8000        ; these must be in RAM




Numberl             ; number to square low byte
Numberh = Numberl+      ; number to square high byte
    .word $FFFF



Squarel             ; square low byte
Squareh = Squarel+1     ; square high byte
    .word $FFFF
```

```
Tempsq              ; temp byte for intermediate result
    .byte $00



    .ORG    8192        ; any address will do Square


    LDA #$00        ; clear A
    STA Squarel     ; clear square low byte
                ; (the high byte gets shifted out)
    LDA Numberl     ; get number low byte
    LDX Numberh     ; get number high byte
    BPL NoNneg      ; if +ve don't negate it
                ; else do a two's complement
    EOR #$FF        ; invert
    SEC         ; +1
    ADC #$00        ; and add it NoNneg:
    STA Tempsq      ; save ABS(number)
    LDX #$08        ; set bit count Nextr2bit:
    ASL Squarel     ; low byte *2
    ROL Squareh     ; high byte *2+carry from low
    ASL A       ; shift number byte
    BCC NoSqadd     ; don't do add if C = 0
    TAY         ; save A
    CLC         ; clear carry for add
    LDA Tempsq      ; get number
    ADC Squarel     ; add number^2 low byte
    STA Squarel     ; save number^2 low byte
    LDA #$00        ; clear A
    ADC Squareh     ; add number^2 high byte
    STA Squareh     ; save number^2 high byte
    TYA         ; get A back NoSqadd:
    DEX         ; decrement bit count
    BNE Nextr2bit   ; go do next bit
    RTS
```

```
;Square Root Calculator
;http://retro.hansotten.nl/6502-sbc/lee-davison-web-site/some-code-bits/#prng
; by Lee Davison
; Calculates the 8 bit root and 9 bit remainder of a 16 bit unsigned integer in
; Numberl/Numberh. The result is always in the range 0 to 255 and is held in
; Root, the remainder is in the range 0 to 511 and is held in Reml/Remh
;
; partial results are held in templ/temph
;
; This routine is the complement to the integer square program.
;
; Destroys A, X registers.



; variables - must be in RAM



Numberl     = $F0       ; number to find square root of low byte
Numberh     = Numberl+1 ; number to find square root of high byte
Reml        = $F2       ; remainder low byte
Remh        = Reml+1    ; remainder high byte
templ       = $F4       ; temp partial low byte
temph       = templ+1   ; temp partial high byte
Root        = $F6       ; square root



    *= $8000        ; can be anywhere, ROM or RAM
```

```
SqRoot


    LDA #$00        ; clear A
    STA Reml        ; clear remainder low byte
    STA Remh        ; clear remainder high byte
    STA Root        ; clear Root
    LDX #$08        ; 8 pairs of bits to do
Loop


    ASL Root        ; Root = Root * 2



    ASL Numberl     ; shift highest bit of number ..
    ROL Numberh     ;
    ROL Reml        ; .. into remainder
    ROL Remh        ;



    ASL Numberl     ; shift highest bit of number ..
    ROL Numberh     ;
    ROL Reml        ; .. into remainder
    ROL Remh        ;



    LDA Root        ; copy Root ..
    STA templ       ; .. to templ
    LDA #$00        ; clear byte
    STA temph       ; clear temp high byte



    SEC         ; +1
    ROL templ       ; temp = temp * 2 + 1
    ROL temph       ;
```

```
    LDA Remh        ; get remainder high byte
    CMP temph       ; comapre with partial high byte
    BCC Next        ; skip sub if remainder high byte smaller



    BNE Subtr       ; do sub if &amp;lt;&amp;gt; (must be
remainder&amp;gt;partial !)



    LDA Reml        ; get remainder low byte
    CMP templ       ; comapre with partial low byte
    BCC Next        ; skip sub if remainder low byte smaller



                ; else remainder&amp;gt;=partial so subtract
then
                ; and add 1 to root. carry is always set here
Subtr



    LDA Reml        ; get remainder low byte
    SBC templ       ; subtract partial low byte
    STA Reml        ; save remainder low byte
    LDA Remh        ; get remainder high byte
    SBC temph       ; subtract partial high byte
    STA Remh        ; save remainder high byte



    INC Root        ; increment Root
Next



    DEX         ; decrement bit pair count
    BNE Loop        ; loop if not all done
```

RTS

```
;Unsigned Integer Division Routines
;https://forums.nesdev.org/viewtopic.php?
f=2&t=11336&fbclid=IwAR3kU9h4klLFp4_8QzsSEXVhicqvjWUWQLqFWR75Q7
rqsyQ3WWRP9UYZFOg




; Unsigned Integer Division Routines
; by Omegamatrix




;Divide by 2
;1 byte, 2 cycles
  lsr




;Divide by 3
;18 bytes, 30 cycles
  sta  temp


  lsr
  adc  #21
  lsr
  adc  temp


  ror
  lsr
  adc  temp
```

```
  ror
  lsr
  adc  temp


  ror
  lsr




;Divide by 4
;2 bytes, 4 cycles
  lsr
  lsr




;Divide by 5
;18 bytes, 30 cycles
  sta  temp


  lsr
  adc  #13
  adc  temp


  ror
  lsr
  lsr
  adc  temp
```

```
  ror
  adc  temp


  ror
  lsr
  lsr




;Divide by 6
;17 bytes, 30 cycles
  lsr
  sta  temp


  lsr
  lsr
  adc  temp


  ror
  lsr
  adc  temp


  ror
  lsr
  adc  temp


  ror
  lsr
```

```
;Divide by 7 (From December '84 Apple Assembly Line)
;15 bytes, 27 cycles
  sta  temp



  lsr
  lsr
  lsr
  adc  temp



  ror
  lsr
  lsr
  adc  temp



  ror
  lsr
  lsr









;Divide by 8
;3 bytes, 6 cycles
  lsr
  lsr
  lsr









;Divide by 9
;17 bytes, 30 cycles
  sta  temp
```

```
  lsr
  lsr
  lsr
  adc  temp


  ror
  adc  temp


  ror
  adc  temp


  ror
  lsr
  lsr
  lsr
```

```
;Divide by 10
;17 bytes, 30 cycles
  lsr
  sta  temp


  lsr
  adc  temp


  ror
  lsr
  lsr
  adc  temp
```

```
  ror
  adc  temp


  ror
  lsr
  lsr




;Divide by 11
;20 bytes, 35 cycles
  sta  temp


  lsr
  lsr
  adc  temp


  ror
  adc  temp


  ror
  adc  temp


  ror
  lsr
  adc  temp


  ror
  lsr
  lsr
  lsr
```

```
;Divide by 12
;17 bytes, 30 cycles
  lsr
  lsr
  sta  temp


  lsr
  adc  temp


  ror
  lsr
  adc  temp


  ror
  lsr
  adc  temp


  ror
  lsr




; Divide by 13
; 21 bytes, 37 cycles
  sta  temp


  lsr
  adc  temp
```

```
  ror
  adc  temp


  ror
  adc  temp


  ror
  lsr
  lsr
  clc
  adc  temp


  ror
  lsr
  lsr
  lsr




;Divide by 14
;1/14 = 1/7 * 1/2
;16 bytes, 29 cycles
  sta  temp


  lsr
  lsr
  lsr
  adc  temp


  ror
  lsr
```

```
  lsr
  adc  temp


  ror
  lsr
  lsr
  lsr




;Divide by 15
;14 bytes, 24 cycles
  sta  temp


  lsr
  adc  #4
  lsr
  lsr
  lsr
  adc  temp


  ror
  lsr
  lsr
  lsr




;Divide by 16
;4 bytes, 8 cycles
  lsr
  lsr
```

```
  lsr
  lsr




;Divide by 17
;18 bytes, 30 cycles
  sta  temp


  lsr
  adc  temp


  ror
  adc  temp


  ror
  adc  temp


  ror
  adc  #0
  lsr
  lsr
  lsr
  lsr




;Divide by 18 = 1/9 * 1/2
;18 bytes, 32 cycles
  sta  temp
```

```
  lsr
  lsr
  lsr
  adc  temp


  ror
  adc  temp


  ror
  adc  temp


  ror
  lsr
  lsr
  lsr
  lsr
```

```
;Divide by 19
;17 bytes, 30 cycles
  sta  temp


  lsr
  adc  temp


  ror
  lsr
  adc  temp


  ror
  adc  temp
```

```
  ror
  lsr
  lsr
  lsr
  lsr
```

```
;Divide by 20
;18 bytes, 32 cycles
  lsr
  lsr
  sta  temp

  lsr
  adc  temp

  ror
  lsr
  lsr
  adc  temp

  ror
  adc  temp

  ror
  lsr
  lsr
```

```
;Divide by 21
;20 bytes, 36 cycles
  sta  temp


  lsr
  adc  temp


  ror
  lsr
  lsr
  lsr
  lsr
  adc  temp


  ror
  adc  temp


  ror
  lsr
  lsr
  lsr
  lsr
```

```
;Divide by 22
;21 bytes, 34 cycles
   lsr
   cmp  #33
   adc  #0
   sta  temp
```

```
   lsr
   adc  temp


   ror
   adc  temp


   ror
   lsr
   adc  temp


   ror
   lsr
   lsr
   lsr
```

```
;Divide by 23
;19 bytes, 34 cycles
  sta  temp


  lsr
  lsr
  lsr
  adc  temp


  ror
  adc  temp


  ror
  lsr
  adc  temp
```

```
  ror
  lsr
  lsr
  lsr
  lsr
```

```
;Divide by 24
;15 bytes, 27 cycles
   lsr
   lsr
   lsr
   sta   temp

   lsr
   lsr
   adc   temp

   ror
   lsr
   adc   temp

   ror
   lsr
```

```
;Divide by 25
;16 bytes, 29 cycles
  sta  temp
```

```
  lsr
  lsr
  lsr
  adc  temp


  ror
  lsr
  adc  temp


  ror
  lsr
  lsr
  lsr
  lsr
```

```
;Divide by 26
;21 bytes, 37 cycles
  lsr
  sta  temp


  lsr
  adc  temp


  ror
  adc  temp


  ror
  adc  temp
```

```
  ror
  lsr
  lsr
  adc  temp


  ror
  lsr
  lsr
  lsr




;Divide by 27
;15 bytes, 27 cycles
  sta  temp


  lsr
  adc  temp


  ror
  lsr
  lsr
  adc  temp


  ror
  lsr
  lsr
  lsr
  lsr
```

```
;Divide by 28
;14 bytes, 24 cycles
  lsr
  lsr
  sta  temp


  lsr
  adc  #2
  lsr
  lsr
  adc  temp


  ror
  lsr
  lsr
```

```
;Divide by 29
;20 bytes, 36 cycles
  sta  temp


  lsr
  lsr
  adc  temp


  ror
  adc  temp


  ror
  lsr
  lsr
```

```
  lsr
  adc  temp


  ror
  lsr
  lsr
  lsr
  lsr







;Divide by 30
;14 bytes, 26 cycles
  sta  temp


  lsr
  lsr
  lsr
  lsr
  sec
  adc  temp


  ror
  lsr
  lsr
  lsr
  lsr







;Divide by 31
;14 bytes, 26 cycles
  sta  temp
```

```
  lsr
  lsr
  lsr
  lsr
  lsr
  adc  temp


  ror
  lsr
  lsr
  lsr
  lsr




;Divide by 32
  lsr
  lsr
  lsr
  lsr
  lsr
```

```
;Unsigned Integer Division Routines
;https://forums.nesdev.org/viewtopic.php?
f=2&t=11336&fbclid=IwAR3kU9h4klLFp4_8QzsSEXVhicqvjWUWQLqFWR75Q7
rqsyQ3WWRP9UYZFOg




;Divide by 6
;1/6 = 1/3 * 1/2
;19 bytes, 32 cycles
  sta  temp


  lsr
  adc  #21
  lsr
  adc  temp


  ror
  lsr
  adc  temp


  ror
  lsr
  adc  temp


  ror
  lsr
  lsr


;Divide by 10
;1/10 = 1/5 * 1/2
```

```
;19 bytes, 32 cycles
  sta  temp


  lsr
  adc  #13
  adc  temp


  ror
  lsr
  lsr
  adc  temp


  ror
  adc  temp


  ror
  lsr
  lsr
  lsr


;Divide by 20
;1/20 = 1/5 * 1/4
;20 bytes, 34 cycles
  sta  temp


  lsr
  adc  #13
  adc  temp


  ror
  lsr
  lsr
  adc  temp
```

```
  ror
  adc  temp


  ror
  lsr
  lsr
  lsr
  lsr


;Divide by 24
;1/24 = 1/3 * 1/8
;21 bytes, 36 cycles
  sta  temp


  lsr
  adc  #21
  lsr
  adc  temp


  ror
  lsr
  adc  temp


  ror
  lsr
  adc  temp


  ror
  lsr
  lsr
  lsr
  lsr
```

```
;Divide by 26
;1/26 = 1/13 * 1/2
;22 bytes, 39 cycles
  sta  temp


  lsr
  adc  temp


  ror
  adc  temp


  ror
  adc  temp


  ror
  lsr
  lsr
  clc
  adc  temp


  ror
  lsr
  lsr
  lsr
  lsr
```

```
;Unsigned Integer Division Routines
;https://forums.nesdev.org/viewtopic.php?
f=2&t=11336&fbclid=IwAR3kU9h4klLFp4_8QzsSEXVhicqvjWUWQLqFWR75Q7
rqsyQ3WWRP9UYZFOg




;Divide by 6
;17 bytes, 30 cycles
  lsr
  sta  temp


  lsr
  lsr
  adc  temp


  ror
  lsr
  adc  temp


  ror
  lsr
  adc  temp


  ror
  lsr


;Divide by 10
;17 bytes, 30 cycles
  lsr
  sta  temp
```

```
  lsr
  adc  temp


  ror
  lsr
  lsr
  adc  temp


  ror
  adc  temp


  ror
  lsr
  lsr


;Divide by 20
;18 bytes, 32 cycles
  lsr
  lsr
  sta  temp


  lsr
  adc  temp


  ror
  lsr
  lsr
  adc  temp


  ror
  adc  temp
```

```
  ror
  lsr
  lsr


;Divide by 24
;15 bytes, 27 cycles
   lsr
   lsr
   lsr
   sta   temp


   lsr
   lsr
   adc   temp


   ror
   lsr
   adc   temp


   ror
   lsr


;Divide by 26
;21 bytes, 37 cycles
  lsr
  sta  temp


  lsr
  adc  temp


  ror
  adc  temp
```

```
  ror
  adc  temp


  ror
  lsr
  lsr
  adc  temp


  ror
  lsr
  lsr
  lsr


;Edit - Divide by 12 has also been superseded!
;Double Edit - Divide by 28 too!


;Divide by 12
;1/12 = 1/3 * 1/4
;20 bytes, 34 cycles
  sta  temp


  lsr
  adc  #21
  lsr
  adc  temp


  ror
  lsr
  adc  temp
```

```
  ror
  lsr
  adc  temp



  ror
  lsr
  lsr
  lsr



;Divide by 28
;1/28 = 1/7 * 1/4
;17 bytes, 31 cycles
  sta  temp



  lsr
  lsr
  lsr
  adc  temp



  ror
  lsr
  lsr
  adc  temp



  ror
  lsr
  lsr
  lsr
  lsr









;new
```

```
;Divide by 12
;17 bytes, 30 cycles
  lsr
  lsr
  sta  temp


  lsr
  adc  temp


  ror
  lsr
  adc  temp


  ror
  lsr
  adc  temp


  ror
  lsr


;Divide by 28
;14 bytes, 24 cycles
  lsr
  lsr
  sta  temp


  lsr
  adc  #2
  lsr
```

```
lsr
adc  temp


ror
lsr
lsr
```