Advanced optimizing

In addition to the tutorials about speedcode and its generation i want to show some other possibilities to save some cycles and thus speed up your code. Therefore i try to give you some triggers on common situations. Feel free to add some more examples.

Branches and conditional code blocks

Branches take 2 cycles if not taken, and 3 cycles if taken. Having this in mind, we can quickly save one cycle by choosing our branch wisely.

```
        ...
        ;some code
        bcs +
        ;return
        rts
+
        ;continue
        ...
```

This can be better written as:

```
        ...
        ;some code
        bcc +
        ;continue
        ...
+
        ;return
        rts
```

So always see that the block that is executed many times is preferred and is not wasting time, while the more expensive things should be moved to the block that happens less often (or even happen outside the loop). Even better, when you can combine branches by putting the decision to the end of a code block and thus save a whole branch:

```
-
        ...
        ;some code
        bcc +
        ;continue
        ...
        jmp -
+
        ;return
        rts
```

The above can happen sometimes, and might be hard to avoid in certain cases, but often a closer look reveals that it could also work like this:

```
-
        ...
        ;some code
        ;continue
        bcs -
        ;return
        rts
```

Also, if you need to load a register depending on some branch, you might be able to save some cycles. Imagine you have the following to load Y depending on the state of the carry:

```
        cmp $1000
        bcs +
        ldy #$00
        jmp ++
+
        ldy #$01
++
```

This can be solved in less cycles and less memory:

```
        ldy #$01
        cmp $1000
        bcs +
        ldy #$00
+

or:
        cmp $1000
        lda #$00
        rol
        tay
or if Y shall either be $80 or $00:
        cmp $1000
        arr #$00
        tay
```

For saving space (not cycles, in fact this is pretty expensive!), BIT is your friend as you can “jump over” a one or two byte command via a BIT instruction. Consider this code:

```
        beq +
        lda #$04
        sta somewhere
        rts
+       lda #$05
        sta somewhere
        rts
```

This can be reduced using a BIT instruction. The opcode for BIT is $2c:

```
        beq +
        lda #$04
        .byte $2c
+       lda #$05
        sta somewhere
        rts
```

This way, the processor will see a bit $05a9 after the lda #$04. The lda #$05 is not executed as it is part of the BIT command. And the BIT command does not change any registers, it only sets some flags. This can be stacked endlessly:

```
        lda #$04
        .byte $2c
foo1    lda #$05
        .byte $2c
foo2    lda #$06
        .byte $2c
foo3    lda #$07
        ...
        sta somewhere
        rts
```

In much the same way, also one byte commands can be “ignored” using the opcode for BIT zeropage, $24.

Determining block number

When doing graphics usually the 8 pixel block restrictions apply, so it is a good thing to break down code to block size and execute blocks of code per block. For that reason you would need the number of full blocks that you handle. There's different approaches of how to get there. Actually all you have to perform is (x1 & $1f8)) - (x2 & $1f8)) / 8 (always assuming that x1 is > x2)

So what you can do is the approach (x1 & $1f8) / 8 - (x2 & $1f8) / 8. As both components are divided by 8, the & $1f8 can be dropped as the bits vanish anyway. But we are limited to 8 bit then:

```
         ldx x1low
         ldy x2low
         lda div8,x
         sec
         sbc div8,y
```

However if x1 and x2 are 9 bit numbers we get into trouble. Imagine x1 is $100 and x2 is $f0? So if we'd just subtract the lowbytes we would end up in doing a $0 - $1e what would result in $e2 blocks what is defenitely wrong. Correct would be in fact to perform $20 - $1e. So here comes the approach to handle also 9 bit subtractions at no extra costs by just putting the shifting to the end:

```
        lda x1low
        ora #$07
        sec
        sbc x2low
        lsr
        lsr
        lsr
```

As you see first of all the lower 3 bits of x1 are all set to avoid an underflow in the lower 3 bits and what would simulate the same as a (x1 - (x2 & $1f8)). However performing an and-operation on the second operand would force us to do the operation beforehand and store the result somewhere. So we first of all turn things around and avoid the underrun on the lowest 3 bits not by masking them out, but by maximizing those bits what gives the same result. When done so we would subtract x2 and finally shift 3 times to divide by 8. Now going through that example with our previous 9 bit values we see that the following happens:

```
$00 | $07 = $07
$07 - $f0 = $17
$17 >> 3 = $02
```

So no matter what lower three bits of x2 would be set, we would be save from an an underrun and end in values from $10 up to $17. Finally when shifting now, all is fine. As the subtract wraps around at the right bit, we end up with a result of $02. However keep in mind that the distance between x1 and x2 must not be greater than $ff.

Alternatives would be: (-(x1 & $f8) + x2) or (-x1 | 7 + x2)

```
        lda x1
        and #$f8
        eor #$ff
        sec
        adc x2
        ...
        lda x1
        eor #$ff
        sec
        ora #$07
        adc x2
```

Indexing and counting

Indexing makes life easier, but implies increment/decrement of the index and checking against some endvalue. Expensive!

Absolute indirect addressing

The absolut inidrect addressing is, what we often want for picking values from a table or for e.g. the screen. But the instruction set of the 6502 only allows us to do the indirect adressing indexed, so usually the index is in our way and we end up in doing stuff like:

```
          sty y+1
          ldy #$00
          lda (screen),y
y         ldy #$00

          ;or

          stx x+1
          ldx #$00
          lda (screen,x)
x         ldx #$00
```

If we now imagine having static or predictable values for X we could forgo on saving the register, but incorporate it into the pointer. For the case of Y this is even more expensive, but in the case of X we can easily do:

```
          ldx #$bf
          lda (<(screen - $bf),x)
```

Though this way of indirect addressing costs 6 cycles it saves the overhead of storing and restoring X. There's no need to set X to zero, but you can just subtract X from the screen-value. To avoid an underrun we therefore take the lowbyte of the result by using '<'. As the 6502 has the $ff-wrap around bug, values will not fetched from above ($ff) or below ($00), so also no need to bother about getting out of bounds.

Comparisons/Faster loops

The compare at the end of a loop influences the carry flag, what can be rather annoying if you calculate in a loop (but can also come handy if you need to clear or set the carry in each round)! So sometimes we are really glad to get rid of it and thus save the compare (plus maybe even a CLC/SEC). Just imagine the following code:

```
        ldy #$18
-
        sta $1000,y
        dey
        cpy #$10
        bne -
```

Wouldn't it be better like this?

```
        lda #start
        sec
        sbc #end
        ;calc delta-y
        tay
        lda #end
        ;add offset beforehand
        sta tgt+1
-
tgt     sta $1000,y
        dey
        bne -
```

Now we can count down to zero, and there is no need for any comparison beforehand, by simply adding the end-value as a fixed offset to the target and counting down the delta of start and end. However be aware of the fact, that a penalty cycle applies if you use LDA and cross a pageboundary (y+#end >= $100). STA will always use 5 cycles.

Using zeropage

Storing to zeropage saves one cycle, compared to a store on a 16-bit address. Also it allows us to store X and Y-register directly what saves an additional TXA/TYA and thus A is not clobbered anymore. So think about it, sometimes it is wise to store some data in zeropage. Imagine the following loop:

```
        ldy #$40
        ldx #$00
-
        txa
        sta $1000,y
        dex
        dex
        dey
        bne -
```

But if there is enough space in zeropage, you'd better do:

```
        ldy #$40
        ldx #$00
-
        stx $80,y
        dex
        dex
        dey
        bne -
```

Even more advantages arise when you place a whole piece of code in the zeropage mostly when selfmanipulating your code. Here's some example that makes this more clear:

```
       ;Example 1
my_y = * + 1
       ldy #$00
       iny
       sty my_y
       ;..do things that clobber y

       ;Example 2
       lda #$00
       sta data
       lda #$10
       sta data+1
data = * + 1
       lda $1000,y
       ...
       lda (data),y
```

First, we can save registers to zeropage with 3 cycles, but can get the value back with only two cycles. So there's one cycles saved whenever we run into scenarios where we need to use a register for more than one purpose. The second part shows that when having code in the zeropage, constructs like the indirect indexed addressing can be omitted if the address is used only once, thus saving again one cycle. Further more one can easily reuse the set up address by referencing the label (data) later on with an indirect indexed opcode. Something you can't do easily when running code outside the zeropage.

Using the stack

PHA and PLA push/pull the accumulator to the stack and decrement/increment the stack-pointer for free. So if we need to sequentially store a bunch of values somewhere, this could be your option:

```
        ;save stack pointer
        tsx
        stx $02
        ;set to our target ($0180)
        ldx #$80
        txs
        lda #$00
        clc
-
        pha
        adc #$10
        bcc -
        ;restore stackpointer
        ldx $02
        txs
```

Further advantage of this method is, that we have an additional register free, as it is not used for an index anymore. But be aware! You have to take into account, that you have to store values top-down, as the stack-pointer decreases on every push. The advantage is, that if an interrupt occurs in

between, it will not trash your values on the stack, as it pushes its 3 bytes (PC + Status) below your current position. All you need to take care of is, that you don't under-run the stack in case of an interrupt (needs 3 bytes, if you do a JSR in the interrupt-handler, another 2 bytes are needed per level), or trash still valid content in the upper part of the stack. For reading out your values from stack you can either use pla but much easier via e.g. lda $0100,x

Counting with steps greater than 1

Later we will discover to do that also by SBX, but there's also another option to do that easily and being able to use LAX features for the index or even function that we walk along

```
count = $20
           ldx #$00
           ldy #$00
-
           stx count,y
           iny
           txa
           sbx #-3
           cpx #$60
           bne -

           ...

.index     lax count
           ...
           do stuff with X and A
           ...
           inc .index + 1
```

As you see the inc .index + 1 will fetch the value from the next location in zeropage on the next turn Thus we have A and X increased by 3 on each round, all done in 9 cycles, and with the option of destroying x later on.

Counting bits

As we are on a 8 bit machine, counting from or to 8 occurs quite often. So why not counting bits?

```
        ;setup counter
        lda #$80
        sta $02
-
        ;do stuff that best use A, X and Y
        lsr $02
        bcc -
        ;restore counter
        ror $02
```

This gets even cooler when you are able to use $02 as some bitmask (for e.g. when drawing lines). When using BMI/BPL or BVS/BVC (need then to test bits with BIT however) you might even count to 1, 2, 6 or 7.

Run length

In unrolled loops the current value of an virtual index can be determined by the code position, thus it is possible to separate the modiyfing part of a loop from the testing part. Let us use an example again to make this more obvious:

```
        lda mask
        sta (bmp),y
        iny
        txa
        sbc dy
        tax
        bcs +
        ;update
+
        lda mask
        sta (bmp),y
        iny
        txa
        sbc dy
        tax
        bcc +
        ;update
+
        ...
```

As you see, there's a nice unrolled loop but for every step we need to reload mask and save/restore our calculation results to the x-register.

If we would now extract the store part we could aggregate the stores:

```
        lda mask
        sta (bmp),y
        iny
        sta (bmp),y
```

And on the other hand also do the calculations without using the x-register, as A is all free now:

```
        sbc dy
        bcs +
        ;update
+
        sbc dy
        bcs +
        ;update
```

```
+
        ...
```

So when all merged together we would end up with the following code:

```
        sbc dy
        bcc one_times
        sbc dy
        bcc two_times
        sbc dy
        bcc three_times
        ...
one_times
        lda mask
        sta (bmp),y
        jmp update
two_times
        lda mask
        sta (bmp),y
        iny
        sta (bmp),y
        jmp update
three_times
        lda mask
        sta (bmp),y
        iny
        sta (bmp),y
        iny
        sta (bmp),y
        jmp update
```

It is obvious that code size restricts this method a bit as the branches don't reach too far. But even then there's situations where it is worth to spend a long branch construction while still saving cycles.

Clobbering registers

Registers are very scarce on the 6502, as we have only the accumulator for arithmetic operations and two index-registers. Running out of registers is very expensive, as we need to save and then restore again registers:

```
        ...
        sta $02
        txa
        sta $1000,y
        lda $02
        dey
        ...
```

The following gets much faster if you run out of registers:

```
        ;before loop, setup y offset
        sty tgt+1
        ...
tgt     stx $1000
        dec tgt+1
        ...
```

This way we avoid clobbering A and we can even get Y free for other use. Although the 6 cycles of the DEC appear expensive, we save 3 + 2 + 3 + 2 cycles + 1 cycle on the unindexed STX now. 5 cycles faster, nice!

The carry

CLC and SEC can make our additions and subtractions twice as expensive, so always keep track of if the carry is set or cleared and if we can reuse it. Also there are some possibilities to set it for free as a nice side-effect from other instructions. Other instructions leave your carry unclobbered but still lead to the same result.

Watch out for BCS and BCC, if you use them within your code, you have the best evidence on if your carry is set.

Example 1:

```
        bcs +
        ;clc
        adc #$10
        sec
+
        ;sec
        sbc #$08
        ...
```

Example 2:

```
       lda #$00
       clc
       adc #$10
       sta $10
       ;clc carry is still clear, as above addition can never overflow
       adc #$10
       sta $11
       ...
```

Also, sometimes an EOR, ORA or AND will just do the same like an ADC or SBC, but without clobbering your carry, and without regarding its current state.

```
        rol $02
        lda $fb
        eor #$80
        sta $fb
        bpl +
        dec $fc
+
        ;carry is still okay here
```

When inside a loop, you can also use the compare at the end of the loop to set/clear your carry automatically:

```
        sec
-
        sbc #$10
        dey
        cpy #$00 ;sets carry again (upcoming bcs shows that clearly)
        bcs -
```

if you need a cleared carry all the time, then do an incrementing loop and branch on clear. Also you might want to have a look at this article: Some words about the ANC opcode

If the carry has not the desired state, one can still circumvent a CLC/SEC beforehand by just taking the state of the carry into account:

```
        ;sec
        adc #$07   ;actually we want to add 8, but carry is set, so 7 is
enough
        ...
        ;clc
        sbc #$07   ;actually we want to subtract 8, but carry is clear, so 7
is enough
```

Another way to circumvent a wrong carry state is to do the following (and if we are lucky the adc does not overflow and we also save the clc):

```
        ;sec
        adc value1
        clc
        sbc value2
        ;gives value1+1-value2-1 = value1-value2
```

When we have to set/clear the carry often due to overflow/underflow of the value, depending on the range of your added/subtracted values, it is smart to shift the value beforehand:

```
        clc
        lda value1
        ora #$80    ;add 128
        sbc #$40
        adc #$60
```

```
        sbc #$40
        adc #$20
        ;... still no under/overflow and last carry state can always be
reused
        eor #$80   ;subtract 128 (can maybe even use and #$7f or even anc to
influence carry!)
```

Sometimes we can substitute a subtraction by a compare. Then we don't even clobber A, but also are not in need to set carry beforehand (but might get the carry set for free):

```
        lda $1000,y
        and #$f8
        cmp xpos       ;substitutes sec + sbc xpos
        bcc +          ;now we have even a reliable state for our carry in
both cases
        ... some code ...
+
```

Additions don't need to happen obviously, but can occur implicitely by indexed opcodes. A small example will explain this better:

```
        lda pos
        clc
        adc offset
        tax
        lda tab,x
        ...
        ;in other words:
        ldx pos
offset = * + 1
        lda tab,x
        ...
        ;zero page indirect y-indexed
offset  = $b0
        ;prerequesite, set up highbyte of pointer
        lda #>tab
        sta offset+1
        ;now read from tab + ypos + offset
        ldy ypos
        lda (offset),y
```

As you see, pos and offset are also added implicitely by the indexing done by the opcode. This might in some situations perform faster and will avoid clobbering the carry.

Last but not least, you might give SBX a try which does not care about the state of the carry on a subtraction.

Use of immediate values

Using immediate values instead of values from memory saves cycles as well. So when inside a loop, think of presetting values before entering the loop, instead of fetching them from mem again and again:

```
        ...
        lda $02
        and mask,x
        sta $1000,y
        ...
```

Now lets save one cycle in the inner loop:

```
        ;before loop
        lda $02
        sta val+1
        ...
val     lda #$00
        and mask,x
        sta $1000,y
        ...
```

If you now even manage to combine the static value with the mask, you could even save 2 additional cycles and simply do a lda mask,x. Also you might load a register with an often used value, and reuse that over the whole loop. This comes handy if you want to apply a fixed mask to a huge load of values when using the SAX command.

Shifting

LSR and ASL is actually a good way to divide/multiply (see also here) by two when handling unsigned numbers, the 2 cycles don't hurt, but might, when doing excessive shifts. When we want to relocate a bit within a byte we can do that from both sides. Imagine we want to get bit 7 down to bit 0:

```
        lda #$80
        lsr
        lsr
        lsr
        lsr
        lsr
        lsr
        lsr
```

Quite some shifts needed, but why not using the possibility of wrap-arounds?

```
        lda #$80
```

```
        asl
        rol
```

Another nice trick to transform a single bit into a new value (good for adding offsets depending on the value of a single bit) offset is the following:

```
        lda xposl  ;load a value
        asr #$01   ;move bit 1 to carry and clear A
        bcc +
        lda #$3f   ;carry is set
+
        adc #stuff ;things will work sane, as offset includes already the
carry
```

As you can see we have now either loaded $00 or $40 (carry!) to A depending on the state of bit 0, that is ideal for e.g. when we want to load from a different bank depending on if a position is odd or even. As you see, the above example is even faster than this (as the shifting always takes 6 cycles, whereas the above example takes 5/6 cycles):

```
        lda xposl
        asr #$01
        ror
        lsr
        adc #stuff ;things will work sane as carry is always clear (upper
bits are masked out)
```

If you want to do the same as above but have to preserve the carry (maybe because you are just preparing to calculate the highbyte and have a carry from the previous lowbyte calculation) then you can use this variant:

```
        lda xposl
        and #$01
        beq +
        lda #$40
+
        adc #stuff ;will now include the carry
```

The given examples show, that asr/arr is nice to combine shifting with masking. So here is another nice example (thanks to Peiselulli) to easily fetch 2 bits from a byte:

```
        lda %10110110
        lsr
        asr #$03*2
```

This will mask out and shift down bits 2 and 3. Note that the mask is applied before shifting, therefor the mask is multiplied by two.

When you intend to copy a certain bit to the carry, you can do that within 2 cycles by comparing. However the bit must be the most significant bit being used:

```
        ldx #$1f
        cpx #$10   ;-> carry is set if bit 4 is set, else it is clear.
        arr #$00   ;A = A & 0, ror, so bit 4 is now bit 7
```

The advantage is, that you can move bits also across registers and are not restricted to the accumulator only.

Jumpcode

If you want to fetch a certain bitpair from a byte (for e.g. fetch the value of a multicolor pixel) you need to invest a variable amount of shifts. To get the shifting more dynamic we can use jumpcode, a trick that not only applies here, but also to various other loops:

```
        lda xposl       ;load some value
        anc #$06        ;either jump 0, 2, 4 or 6 bytes far, clears carry to
force upcoming branch
        sta .jt1+1      ;setup jump
        lda (zp),y      ;load value to be shifted
.jt1    bcc *+2         ;jump into code with right offset
        ;x = 0
        lsr
        lsr
        ;x = 2
        lsr
        lsr
        ;x = 4
        lsr
        lsr
        ;x = 6
        and #$03        ;finally mask out desired bits
```

Looks like a bunch of code, but if you imagine that the equivalent code would be the following, you see that the overhead setting up the jump is nearly the same, not to mention the saved loop-overhead of another 5 cycles per step if you use jumpcode:

```
        lda xposl       ;load some value
        and #$06        ;mask out bits
        eor #$06
        tax
        lda (zp),y      ;load value to be shifted
        cpx #$00
        beq +
-
        asl
        asl
        dex
        bne -
+
```

```
        and #$03
```

If you want to see other examples of jumpcode, have a look at the code presented in Filling the vectors, where it is also used to set the start and endpoint of an unrolled loop.

Even more fun is doing jumpcode via an indirect jump. Imagine you have an unrolled loop of speedcode and want to find out the point where to enter your speedcode depening on an index. Usually the size of one loop is not a power of two and thus things get complicated, even worse when the speedcode segments we jump to have a variable size. Here comes the solution to seize the pain:

```
;first, set up an aligned table of pointers into our speedcode
!align 255,0
dest
        !word speedcode_entry1, speedcode_entry2, speedcode_entry3 ...
enter
        tya
        asl ;shift by two, we use pointers
        sta jump+1
jump    jmp (dest)
        ;this way we simulate a jmp ($xxxx),y where the index may range from
$00..$7f
```

If the targets to be reached are all within the same page, then of course a normal jump with manipulated lowbyte would do as well and save 2 cycles on teh jump. More examples on this can be found here: dispatch_on_a_byte

Combining bits / Substitute logical operations

Mostly with 4×4 effects, but also in other cases you might wish to combine two numbers into a single byte, like e.g. high- and lownibble. You can usually do that like:

```
        lda lownibbles,x
        ora highnibbles,x
        sta target
```

This will for e.g. merge a $c0 and $03 to $C3

This can however also been done the other way round by using an AND operation. Therefore just the unused bits have to be set to 1 instead of 0 So we would then result in for e.g. $cf and $f3 as high- and lownibble, if we now combine them by AND we also get $c3 as result. And then, you might think? Well, there's lots of illegal opcodes that include and operations, but just a few with ora/eor. Most of all, now we can make excessive use of the SAX command and store/manipulate low/highnibbles in A and X seperatedly. So the above code would now be:

```
        lda lownibbles,y
        ldx highnibble,y
```

```
        sax target
```

So far we still use the same amount of cycles, but we are now able to reuse either X or A for the next combinations. In case of using ORA we would first need to mask out the unwanted nibble again to do so. This comes in handy for things like texture mappers, and has been used in the 50fps sphere mapper in coma light 13.

On other occasions you want to mask out for e.g. the lower 3 bits to act as a counter going from 0 to 7 or 7 to 0. Usually when you want the counter inverted this looks like:

```
        lda x1
        and #$07
        eor #$07
        tax
```

Though it can take more cycles, this method might be handy if you can reuse values and/or have the carry already set:

```
        lda x1
        eor #$ff        ;tay + iny gives you for e.g. -val as an extra
        and #$07
        tax
        ;or

        lda #$ff        ;would also work with A = 0 and carry cleared, so we
could save on a few opcodes if so
        sbc x1          ;tay + iny gives you for e.g. -val as an extra
        and #$07
        tax
        ;or if a = $ff
        ;thanks to Kabuto to point that use of lax out in a comment on csdb
        lax #7          ;X and A = 7
        eor x1          ;A = x1 eor 7
        sbx #$00        ;X = A and 7
        ;or if you go for the distance between x1 and x2
        lda x1
        ora #$07        ;(x1 | 7 - x2) / 8 gives you for e.g. the number of
blocks (assumed that x2 < x1)
        sbc x2
        tax
```

If you want to clear certain bits and are using a add/subtract operation anyway, you actually can combine both if the bit to be cleared is set before in any case:

```
        bmi +
        ;...
+
        ;bit 7 is set, clear it
        and #$7f
        sbc #$01
        ;can also be:
```

```
        bmi +
        ;...
+
        sbc #$81 ;-> clears Bit 7 and subtracts 1
```

In the same way this method can also be used to set bits (for e.g. with adc #$81) or to toggle bits.

Illegal opcodes

Now let me show you some nice situations where illegal opcodes can save you a few cycles by combining some mnemonics in a single command. Note that the examples i give are not the only situations where you can make use of illegal opcodes, but they might give you some hint on how they can be used. Also you might have noticed that i used some of the illegal opcodes in my previous examples, so here we go.

LAX

Loads A and X with the same value. Ideal if you manipulate the original value, but later on need the value again. Instead of loading it again you can either transfer it again from the other register, or combine A and X again with another illegal opcode.

```
        lax $1000,y  ;load A and X with value from $1000,y
        eor #$80     ;manipulate A
        sta ($fd),y  ;store A
        lda #$f8     ;load mask
        sax jump+1   ;store A & X
```

Also one could do:

```
        lax $1000,y  ;load A and X with value from $1000,y
        eor #$80     ;manipulate A
        sta ($fd),y  ;store A
        txa          ;fetch value again
        eor #$40     ;manipulate
        sta ($fb),y  ;store
```

If you can afford clobbering A you can also load X with additional addressing modes (remember that ldx ($xx),y is not available) like:

```
        lax ($fb),y
        ;... instead of
        lda ($fb),y
        tax
```

Even more fancy shit can be done by lax ($xx,x), here you can implement a lookuptable that needs the previous value of x as input. So basically you can do any x = f(x); term.

Actually you can use LAX also with an immediate value, but it behaves a bit unstable regarding the given immediate value. However when simply doing an LAX #$00 you are fine.

SAX/SHA

This opcode is ideal to setup a permanent mask and store values combined with that mask:

```
        ldx #$aa     ;setup mask
        lda $1000,y  ;load A
        sax $80,y    ;store A & $aa
```

Also there's a nice example of writing out a row of numbers faster than you might think:

```
        ;write values form 0 .. 7 to screen
        ldx #$0e
        lda #$01
        sax $0400    ;write $01 & $0e == 0
        sta $0401    ;write $01
        lda #$03
        sax $0402    ;write $03 & $0e == 2
        sta $0403    ;write $03
        lda #$05
        sax $0404    ;write $05 & $0e == 4
        sta $0405    ;write $05
        lda #$07
        sax $0406    ;write $07 & $0e == 6
        sta $0407    ;write $07
```

as you see, this wastes just one byte more than an unrolled loop as in the upcoming example, but saves 2 cycles on every second byte written.

This trick also helps when you need to switch 8 sprite pointers in a line. Usually one could just set up 2 different pointers at two different screens and switch 8 sprite pointers via $d018. But this is not applicable if your effect renders stuff into the screen or if you are doing even double buffering with screens. Here you have to fall back to writing 8 new sprite pointers in less then 44 cycles (63-19), but then also cope with possible jitter that is added. Preloading registers will then only help if you have a stable enough irq position, for e.g. achieved by a double irq. Here this fast writing of 8 values helps.

The only thing to take care is, that #sprites is an even number (for odd numbers the sax and sta statements need to be swapped and y should be used for writing the last value). Now we are able to write 8 sprite pointers in 38 cycles.

```
        ldx #$00
        stx $0400
        inx
        stx $0401
        inx
        stx $0402
        inx
```

```
        stx $0403
        inx
        stx $0404
        inx
        stx $0405
        inx
        stx $0406
        inx
        stx $0407
```

An y-index version of *SAX* exists in the illegal opcode *SHA*. However it also adds the highbyte+1 of the used address as a mask to the value written. So in most cases you are restricted to certain destination addresses.

SHX/SHY

When storing to zeropage you can also store the y- and x-register with an index in a fast and comfortable way. But often you will need the zeropage for other things. Sadly the instruction set of the 6510 is not orthogonal and thus this features are not available for 16 bit addresses. You can however workaround that nuisance by using SHX or SHY, but have to cope with the H component in it, as the stored values are anded with the highbyte of the destination address + 1. So most of the time you might want to store to $fexx to not run into any problems. In case you have to apply an additional static mask, or if you just need certain bits of the stored values, you can of course choose a different address. If you start crossing a page with the index, the behaviour of this opcode changes radically. In those cases the Y-value becomes the highbyte of the address the values is stored at.

Want some example?

```
sin_p   = $02
ztab    = $fe00
        lax (sin_p),y       ;load position in ztab
        shy ztab2,x         ;store line num in ztab
        iny                 ;next line
        ...
```

ASR

Whenever you need to shift and influence the carry afterwards, you can use ASR for that, and if you even need to apply an and-mask beforehand, you are extra lucky and can do 3 commands by that:

```
        asr #$fe     ;-> A & $fe = $fe -> lsr -> carry is cleared as bit 0
was not set before lsr
```

... same as ...

```
        and #$ff
        lsr
```

```
        clc
```

ARR

ARR ands the accumulator with an immediate value and then rotates the content right. The resulting carry is however not influenced by the LSB as expected from a normal rotate. The Carry and the state of the overflow-flag depend on the state of bit 6 and 7 before the rotate occurs, but after the and-operation has happened, and will be set like shown in the following table (thanks to doynax for correcting me):

Bit 7	Bit 6	Carry	Overflow
0	0	0	0
0	1	0	1
1	0	1	1
1	1	1	0

So ARR is quite similar to ASR and is perfect for rotating 16 bit stuff:

```
        lda #>addr
        lsr
        sta $fc
        arr #$00  ;A = A & $00 -> ror A
        sta $fb
```

... is the same as ...

```
        lda #>addr
        lsr
        sta $fc
        lda #$00  ;set to #$01 if you want to leave with a set carry
        ror
        sta $fb
```

Note: When using ARR value #$00 or #$80 do the trick to influence the state of the carry after operation, but the later only if A has bit 7 set as well, so be careful here). However this uncommon behaviour enables another trick. Due to the fact that the carry resembles the state of bit 7 after ARR is executed, one can continuously shift in zeroes or ones into a byte:

```
        lda #$80
        sec
        arr #$ff ; -> A = $c0 -> sec
        arr #$ff ; -> A = $e0 -> sec
        arr #$ff ; -> A = $f0 -> sec
        ...
        lda #$7f
        clc
        arr #$ff ; -> A = $3f -> clc
        arr #$ff ; -> A = $1f -> clc
```

```
        arr #$ff ; -> A = $0f -> clc
```

SBX

Finally i found a good use for the SBX command. Imagine you have a byte that is divided into two nibbles (just what you often use in 4×4 effects), now you want to decrement each nibble, but when the lownibble underflows, this will decrement the highnibble as well, here the sbx command can help to find out about that special case:

```
         lda $0400,y    ;load value
         ldx #$0f       ;setup mask
         sbx #$00       ;check if low nibble underflows -> X = A & $0f
         bne +          ;all fine, decrement both nibbles the cheap way,
carry is set! \o/
         sbc #$f0       ;do wrap around by hand
         sec
+
         sbc #$11       ;decrement both nibbles, carry is set already by sbx
```

... can be substituted by ...

```
         lda #$0f       ;set up mask beforehand, can be reused for each turn
         sta $02
         lda $0400,y
         bit $02        ;apply mask without destroying A
         bne +
         clc
         adc #$10
+
         sec            ;we need to set carry :-(
         sbc #$11
```

A second case in which to use SBX is in combination with LAX, for example when doing:

```
         lda $02
         clc
         adc #$08
         tax
```

that can be easily sustituted by:

```
         lax $02    ;A = X = M [$02]
         sbx #$f8   ;X = (A & X) - -8
```

Even multiple subtractions can be made if A stays untouched, and it is also sufficient if A is $ff to disable the AND component of the opcode:

```
         lda #$ff
```

```
        ldx val
        sbx #$07
        ... some code that does not clobber A,X
        sbx #$08
        ...
```

That also means, that we can easily implement a counter in X:

```
        txa
        clc
        adc #value
        tax
        ;can now be:
        txa          ;A = X
        sbx #-value  ;X = A & X -> X = X and then X = X - - value -> X = X
+ value
        ;voila, new value is in X, all done in 4 cycles
        ;or even 2 cycles if A stays $ff as in the example above:
        lda #$ff
        sbx #-value
        ...
        sbx #-value
        ...
```

So we saved 4 cycles here, as the state of the carry is of no interest for the subtract done by SBX, what is one of its big advantages. Thus we could also fake an ADD or SUB with that command. The and-operation is not needed here, but does not harm. If there's use for it, just let A or X be loaded with the right value for the and-mask.

Another trick that makes use of the SBX command is the negation of a 16 bit number (thanks to andym00!):

```
        lax #$00 ;should be save, as #$00 is loaded
        sbx #lo  ;sets carry automatically for upcoming sbc
        sbc #hi
```

One might also think of extending this trick to negate two 8 bit numbers (A, X) at a time.

Furthermore, the SBX command can also be used to apply a mask to an index easily:

```
        ldx #$03      ;mask
        lda val1      ;load value
        sbx #$00      ;mask out lower 4 bits -> X
        lsr           ;A is untouched, so we can continue doing stuff with
A
        lsr
        sta val1
        lda colors,x  ;fetch color from table
        ;instead of (takes 3 cycles more)
        lda val1
        and #$03
```

```
        tax           ;setup index
        lsr val1      ;A is clobbered, so shift direct
        lsr val1
        lda colors,x
```

The described case makes it easy to decode 4 multicolor pixelpairs by always setting up an index from the lowest two bits and fetching the appropriate color from a previously set up table.

In certain cases sbx might even help out to eor X with a value:

```
        ldx #$07
        txa         ;make A = X so that and-component of sbx does not
destroy X
        sbx #$ff    ;-> X = X eor $ff
```

This would also work with other values, for e.g. with $80 to toggle the MSB of X.

DCP/ISC

Thanks go to LHS/Ancients Pledge Inc./Padua for the upcoming example and hints on this illegal opcode:

```
x1     !byte $7
x2     !byte $1a

;an effect
-
       dec x2
       lda x2
       cmp x1
       bne -

can be written as

;an effect
-
       lda x1
       dcp x2    ;decrements x2 and compares x2 to A
       bne -
```

Another good use can be made if you want to do a inc/dec ($xx),y what is actually not available. So here isc/dcp ($xx),y will help you out, as it is also available for the indirect y adressing mode.

f.e.:

```
ldy #..
lda (zp),y
clc
adc #..
```

```
sta (zp),y
bcc +
iny
isc (zp),y
+
```

or

```
ldy #..
lda (zp),y
sec
sbc #..
sta (zp),y
bcs +
iny
dcp (zp),y
+
```

For decrementing a 16 bit pointer it is also of good use:

```
       lda #$ff
       dcp ptr
       bne *+4
       dec ptr+1
       ;carry is set always for free \o/
```

Under certain circumstances the command can be also misused to decrement a value and set/clear the carry for free while doing so.

```
         sec
         lda scr
         sbc #$28
         sta scr
         lda #$bf
         bcs +
         dcp scr+1         ;sets carry for free
+
         adc bmp
         sta bmp
         lda bmp+1
         sbc #$01
         sta bmp+1
         ;... or
         lda dst
         sbc #$08
         sta dst
         bcs *+4
         dcp dst+1
         ;sets carry for free as long as dst+1 is <= $f8
```

So here the carry is set for free as long as the content of scr+1 is lower than $bf.

More examples, (also using ISC) can be found here.

SRE/SLO

SRE shifts the content of a memory location to the right and eors the content with A, while SLO shifts to the left and does an OR instead of EOR.

So this is nice to combine the previous described 8 bit counter with for e.g. setting pixels:

```
        lda #$80
        sta pix

        ...

        lda (zp),y
        sre pix            ;shift mask one to the right and eor mask with A
        bcs advance_column ;did the counter under-run? so advance column
        sta (zp),y

        ...

advance_column
        ror pix            ;reset counter

        lda zp             ;advance column
        ;clc               ;is still clear
        adc #$08
        sta zp
        bcc +
        inc zp+1
+
        lda (zp),y
        ora #$80           ;set first pixel
        sta (zp),y
```

DOP/TOP

For saving space (not cycles, in fact this is pretty expensive!), BIT is your friend as you can “jump over” a one or two byte command via a BIT instruction. Consider this code:

```
        beq +
        lda #$04
        sta somewhere
        rts
+       lda #$05
```

```
        sta somewhere
        rts
```

This can be reduced using a BIT instruction. The opcode for BIT is $2c:

```
        beq +
        lda #$04
        .byte $2c
+       lda #$05
        sta somewhere
        rts
```

This way, the processor will see a bit $05a9 after the lda #$04. The lda #$05 is not executed as it is part of the BIT command. And the BIT command does not change any registers, it only sets some flags. This can be stacked endlessly:

```
        lda #$04
        .byte $2c
foo1    lda #$05
        .byte $2c
foo2    lda #$06
        .byte $2c
foo3    lda #$07
        ...
        sta somewhere
        rts
```

In much the same way, also one byte commands can be “ignored” using the opcode for BIT zeropage, $24.

Alternatively one can also use the illegal opcodes DOP and TOP to skip bytes. The advantage of those is, that flags stay untouched.

```
                bmi .lz_short           ;continue with y = $ff
.lz_far
                eor #$ff
                tay                     ;y = - a - 1

                lda $beef,x
                inx
                bne .lz_join
                jsr .lz_next_sector
                top                     ;one could also branch/jump to skip
the ldy #$ff
                                        ;but this way only one byte is needed
.lz_short
                ldy #$ff
.lz_join
```

Penalty cycles

Those cycles can be a pain in the arse when doing cycle exact timing, but they can also steal us cycles on other occasions. So let us recall on when such an additional cycle is consumed:

- when our index + address crosses a page (read operations)
- branching over a page boundary

So to avoid wasting lots of cycles we need to align tables properly, usually to a page boundary to avoid an overflow on the index. If our code crosses a page we should avoid placing a loop at that edge, as else one penalty cycle is consumed on branching back to the beginning of the loop:

```
.C:0ffc   A2 00      LDX #$00
.C:0ffe   9D 00 20   LDA $2080,X ;needs 1 cycle extra if X >= $80
.C:1001   E8         INX
.C:1002   D0 FA      BNE $0FFE   ;needs 4 cycles if branch is taken
```

Best is to add some warnings around important loops, so that you get a notice when you loop is badly aligned. For ACME for e.g. you could do that by comparing the highbytes of the loop's start- and end-address:

```
         ldx #$00
loop1
         sta $2000,x
         inx
         bne loop1
!if >* != >loop1 { !warn "loop1 crosses page!" }
```

Sometimes we are happy if we manage to have a branch always condition and thus save one byte of code. But be careful to not waste valueable cycles if this branch is going to cross a page boundary. In that case jmp is cheaper but wastes more bytes.

Forming terms

Sometimes forming terms helps in creating more efficient code. Imagine you have a term A - B where you want to reuse B afterwards, you might do:

```
          lda $dead
          sec
          sbc $beef
          sta $02
          lda $beef
          sta $03
          ;20 cycles
```

By forming the term to - B + A you will result in the same value, but now as the order is changed the

reloading of B can be omitted. There's also the possibility to use LAX and TXA if you can allow for using X.

```
          lda $beef
          sta $03
          sec
          eor #$ff
          adc $dead
          sta $02
          ;18 cycles
```

So always try to form the term into something new and see if it performs better this way. So just remember the simple mathematic laws.

Now also think of that classical negation term:

```
          lda num
          eor #$ff
          clc
          adc #$01
          sta neg
```

Depending on what you have in register A, you can express it in many differnet ways:

```
          ;a = $ff; carry set
          eor num
          adc #$00
          sta neg
          ;a = $00; carry set;
          sbc num
          sta neg
          ;a = $ff; carry clear
          adc num
          eor #$ff
          sta neg
          ;a = $00; carry clear;
          adc #$01
          sbc num
          sta neg
```

There are of course also other expressions possible, just ponder a while about the term.

Running out of registers

Under certain circumstances you can use the stack-pointer as a 4th register. That is, when the code segment using the txs/tsx is not called as a subroutine and returning at some point. Also no data and code must be located inside the stack, as the next IRQ will write three bytes at the stack-pointers position and thus trash data there. But as long as those prerequisites are matched, SP can be used as

an additional register to stow away data.

```
        lax (table),y
        txs
        ldx #$07
        sbx #$00
        ldy pos
        ...
        ...
        tsx ;fetch value from table again
```

Misc stuff

Postponing branches

Did you ever run into such a situation where you need to take a decision within your code but are in need of the registers for other purposes. Under certain circumstances you can perfectly work around that problem by compare in time and branching later on. This avoids reloading registers for the sake of comparing and thus wasting cycles:

```
-
        ;... some code
        ldx counter
        cpx #100         ;make decision here as long as X still contains
counter -> sets/clears carry
        ldy color1,x
        lda color2,x
        tax
        lda color3
        bcc -            ;carry state still untouched, so we can still
branch
```

Incrementing related pointers

Imagine you have a pointer into a bitmap and a corresponding screen. While the screenpointer increments/decrements by 1 the bitmappointer does so by 8. So usually one could do so by:

```
        ;increment
        lda bmp
        clc
        adc #$08
        sta bmp
        bcc *+4
        inc bmp+1
        inc scr
```

```
        bne *+4
        inc scr+1
        ;decrement
        lda bmp
        sec
        sbc #$08
        sta bmp
        bcs *+4
        dec bmp+1
        lda #$ff
        dcp scr
        bne *+4
        dec scr+1
```

This needs bestcase 21 cycles. But actually the check on the overflow of the lowbyte from the screenpointer is only necessary when the bitmappointer's lowbyte also overflows:

```
        lda bmp
        clc
        adc #$08
        sta bmp
        inc scr
        bcc ++       ;bitmap did not overrun, finished
        bne +        ;screen lowbyte did overrun?
        inc scr+1
+
        isc bmp+1    ;force carry clear for free (a = 0 -> a - bmp+1 will
allways underflow)
++
```

Quite some brainfuck, but only needs 18 cycles bestcase. And here's the decrement case that saves another 2 cycles compared to the above example:

```
        lda bmp
        sec
        sbc #$08
        sta bmp
        bcs +
        dcp bmp+1     ;forces carry set for free as long as bmp+1 is <= $f8
(the underflow result in A from above subtraction)
        lda scr
        bne +
        dec scr+1
+
        dec scr
```

Playing with the programcounter's wraparound

When you place code in the zeropage you might run out of space, but when placing code at the

beginning of the zeropage and at the end of the ram above $ff80 you can easily branch back and forth and across the wrap-around. Means one can branch with a beq *-$70 back from zeropage to $ffxx and back to zeropage when branching into the other direction. Sometimes that is a good thing when you can avoid far branches done with a combination of a branch and jump.

So you can easily do things like the following iif your assembler supports it by labels, if not, you have to do the branches manually with a *+$xx or *-$xx:

```
fff3   90 18      BCC $000D       ;directly go to zeropage, no need for a
far jump, as this wraps around
...
002c   30 87      BMI $FFB5       ;same here
```

Reuse of intrinsic information

When your code handles conditions, the code branched to holds some intrinsic information, this can be used to restore values and registers or to save other overhead. The upcoming example unrolls a loop that sets bytes in mem from y to 7:

```
         ldx ymul3,y
         stx .bra+1
         ldy #$07
.bra     bpl *
         sta (dst),y
         dey
         sta (dst),y
         dey
         sta (dst),y
         dey
         sta (dst),y
         dey
         sta (dst),y
         dey
         sta (dst),y
         dey
         sta (dst),y
         dey

ymul3
         !byte $00,$03,$06,$09,$0c,$0f,$12
```

As you see, the branch destination still contains the information about Y (Y * 3) that is then again restored by the right amount of dey. So this loop leaves Y unclobbered. When dispatching on a byte the value being dispatched on can be safely assumed and for e.g. set in the corresponding code segment with ldy #dispatch_val.

Executable parameters

Sometimes we are happy and a parameter, be it in a register or in an other opcode happens to just be the opcode that we want to execute, either directly or in another case when branching. Thus the jmp $dd0c trick happens to work, but also other scenarios could be possible:

```
.C:0010  68          PLA
.C:0011  E9 00       SBC #$00
.C:0013  01 06       BCS $001B
.C:0015  69 00       ADC #$00
.C:0017  48          PHA
.C:0018  29 0F       AND #$0F
.C:001a  85 48       STA $48
```

As A is modified in the one case a common point where both paths merge again with a PHA is impossible. But by storing A at a wise address ($48 = opcode PHA) we can successful merge both parts at $001c without further awkwardness like an additional branch/jump. Thanks a lot to lft for pointing me to this!

HAPPY OPTIMIZING!

Bitbreaker/Oxyron^Nuance