The 6502 Ain’t That Bad

by Kent Dickey

1. Initial Response

Jesús Arias writes in https://www.ele.uva.es/~jesus/onthe6502.pdf “On the 6502: A brilliant or sloppy design?” about how the 6502 is overrated and how the Z80 is underrated.

Jesús has done a lot of 6502 work, so he is knowledgeable on the subject, but he’s overlooking historical factors which mitigate many of his complaints.

In 1976, microprocessors were new, and it wasn’t even clear there was a significant market. What the 6502 clearly tried to do was to enable the lowest system cost while also being reasonably efficient. For the first part, the chip needed to be cheap, and it needed to not require a lot of expense in the system. For the second part, the 6502 generally is doing a useful operation on the bus almost every cycle (the author notes it wastes only about 9% of the cycles).

The author is criticizing 6502 performance compared to what could have been done. Nowadays, system performance is a key metric, since that’s a differentiator. But it really wasn’t a big concern in 1976—designing something that was useful was more important than outright performance.

Plus, 1976 had primitive design tools. Effectively, you draw polygons using crayons, and take a photograph, and that’s how you make masks. There were almost no tools to help optimize or even check your work. The 6502’s PLA design is a clever labor reduction suited to this era: not for performance or area, but for labor costs.

2. Z80 vs 6502 comparison

The 6502 came out in 1976 at 1MHz. The 6502 is so simple to put in a system, that Steve Wozniak was able to create a complete computer, with sound, color graphics, etc. using one board of basically just TTL and (RAM and ROM) chips. The 6502 has such generous timing margins that he could sneak in video fetches from the system RAM during the first phase of the clock, which provides memory refresh as well.

The Z80 also came out in 1976, but at 2MHz. As best as I can tell, the Z80 doesn't offer a 4MHz version until 1981. Z80 system design made the CPU speed easier to change, so Z80 systems moved to the new speeds as they became available. The Apple II and Commodore could not easily upgrade speeds due to software relying on the 1MHz speed.

3. 6502 criticisms

Jesús argues about some bad decisions in the 6502. To me, its best to think of the 6502 as quirky, but well documented, and just accept dummy cycles, the weird B flag, ADC and no ADD, the JMP (0FFF) bug, etc.

The lack of 3-state address bus is not an issue for many system designs which buffer the CPU address before connecting to all the system peripherals. Address decode can be faster with no tristate since no qualification is needed, and no bus holders are needed.

And BCD is useful, as the Sieve example will show. BCD gets the carry flag correct, and that's all that's usually needed. It would be an unusual case where the N or Z flag would be needed by code after a BCD ADC/SBC so it is not really an issue.

The way the 6502 uses carry is standard to many CPUs (where SBC requires C=1 to do a simple subtract), such as Arm, and is less hardware (x86 and Z80 must invert the CF bit before doing subtract, and must invert the carry out of the ALU before writing CF. Since x86 and Z80 share a history, this to me is an oddball thing that they share). The 6502 (and other CPUs) implement subtract by simply inverting each bit of one operand.

4. Sieve of Eratosthenes

The Sieve of Eratosthenes was selected as a benchmark for 8-bit CPUs. The 6502 has a performance disadvantage for operating on data of more than 256 bytes in an inner loop, so a Sieve should give a Z80 a slight edge. But the author chose an encoding of just 256 bytes for the Sieve array, but using all of the bits in the byte to have 2048 bits. Jesús chose to find primes from 2 through 2048 and print them out in decimal.

There are many algorithmic variations on a Sieve. The Byte Benchmark version stores only odd numbers in the array, so a 8192 byte array can count to 16384 (using a byte as a flag), and it skips the cross-off-by-2 step. The Byte Benchmark version does not consider printing the values, it just counts the number in the indicated range. Optimizations allow skipping a lot of work: when crossing off numbers, you do not need to cross off any more once you've found a prime >= sqrt(N), which would be 46 in this case. This saves a great deal of work since the first entry to be crossed off for prime P is P*P. By tracking P*P as well as P, this can also save a lot of work by starting the crossing-off at P*P. This helps less so for a limit of 2048, which is relatively small, and less so for this instance due to the use of bits instead of bytes. And a segmented Sieve can be a good fit for 8-bit CPUs by doing the work in two parts: a simple Sieve to find primes through sqrt(N), and then stepping through segments of 256 bytes or less to find the remaining primes.

Let's assume significant algorithmic changes are off the table. We'll stick to storing 2048 bits in 256 bytes, and need to print out the primes in decimal.

4.1. Original 6502 code

Here is the code from onthe6502.pdf, changed to Merlin format and Apple II addresses (basically, remove : from the end of labels, use $1000 for the code, $2000 for the array):

```
tmp1        equ    $00
tmp2        equ    $01
number      equ    $02      ; and $03
index       equ    $04      ; and $05
array       equ    $2000

            org    $1000
sieve       ldy    #0
            lda    #$ff
l1          sta    array,y
            iny
            bne    l1

            sty    number+1       ; Start with number=2
            lda    #2
            sta    number

mbuc        lda    number
            sta    tmp1
            lda    number+1
            sta    tmp2
            lsr    tmp2           ; y = number/8
            ror    tmp1
            lsr    tmp2
            ror    tmp1
            lsr    tmp2
            ror    tmp1
            ldy    tmp1

            lda    number         ; A = 1 << (number & 7)
            and    #7
            tax
            lda    #1
            cpx    #0
            beq    l3
l2          asl
```

```
            dex
            bne     l2
l3          and     array,y         ; check bit
            bne     l35             ; not prime
            jmp     nxn
l35         lda     number          ; number is prime. print it
            sta     tmp1
            lda     number+1
            sta     tmp2            ; tmp1,tmp2: data to be printed
            ldy     #0
prn1        ;--------- divide tmp1,tmp2 by 10. Remainder in A
            ldx     #16
            lda     #0
dv1         asl     tmp1
            rol     tmp2
            rol
            cmp     #10
            bcc     dv2
            sbc     #10
            inc     tmp1
dv2         dex
            bne     dv1
            ;-------------
            clc
            adc     #$b0
            pha
            iny
            lda     tmp1
            ora     tmp2
            bne     prn1
            ;-------------
prn2        pla
            jsr     cout
            dey
            bne     prn2
            lda     #$a0
            jsr     cout
            ;------------- Mark every multiple of number as not prime
            lda     number          ; index=number
            sta     index
            lda     number+1
            sta     index+1
buc2        clc                     ; index+=number
            lda     index
            adc     number
```

```
                sta     index
                sta     tmp1
                lda     index+1
                adc     number+1
                sta     index+1
                sta     tmp2

                lda     #8              ; if (index >= $800) break
                cmp     index+1
                bcc     nxn

                lsr     tmp2            ; y = index/8
                ror     tmp1
                lsr     tmp2
                ror     tmp1
                lsr     tmp2
                ror     tmp1
                ldy     tmp1

                lda     index           ; A = ~(1 << (number & 7))
                and     #7
                tax
                lda     #1
                cpx     #0
                beq     l7
l6              asl
                dex
                bne     l6
l7              eor     #$ff

                and     array,y         ; mark the bit
                sta     array,y
                jmp     buc2

nxn             inc     number          ; number++
                bne     l5
                inc     number+1
l5              lda     number+1        ; if (number & 0x7ff) != 0 continue
                cmp     #8
                beq     theend
                jmp     mbuc

theend          rts

cout            rts
```

The basic algorithm is to have 256 bytes in array, initialized to all 1's, and treat the array as 2048 bits. Start "Number" at 2, and index into Array to get that bit with index=Number >> 3, bit = Number & 7. If that bit is set in Array, it's a prime, print out "number" in decimal, and then cross off all multiples of "Number" in the array by clearing those bits. Increment Number until it's more than $800, and then stop.

Not counting the RTS at "theend", this code takes 1181744 cycles. Changing the RTS at "cout" to "JMP $FDED" allows output on an Apple II.

Unfortunately, this code has a bug. The output starts with (this was pointed out by John Brooks, I missed this):

2 3 5 11 23 29 41 59 71 83 89 101 113 131 …

Which is missing 7, 13, 19, etc. What's happening is the cross-off code is detecting the end of the array improperly with: "LDA #8"; "CMP index+1"; "BCC NXN". This branches to NXN (and stops crossing off multiples of this number) when 8 < (index+1). This occurs when the bit offset is $900 or higher—which is too high, it should stop at $800. When crossing off 3, it wraps around from bit 2048 back to 0 and crosses off 1, 4, 7, 10, etc. This takes more time as well, since it's crossing off more numbers. A fix (which matches how most people think about CMP and BCC/BCS) is to swap the LDA and CMP arguments and do: "LDA index+1"; "CMP #8"; "BCS NXN".

With this fix, the cycles drops to 1162093, which is a 1.7% improvement. But we can do better.

4.2. "Improv1" code

It's important with 6502 to try to keep working data in the accumulator. "ASL" of the accumulator takes 2 clocks, but "ASL $02" to a zero-page location takes 5 clocks. There are places in the code where this would be helpful: it takes less code and is faster. Another 6502 trick is to use small lookup tables rather than loops. Rather than performing "1 << shift" shifting by one bit "shift" times, just lookup in an 8-entry table. We need 1 << shift for shift from 0…7, and ~(1 << shift) for shift from 0…7. This is just 16 bytes of tables, so it's a definite win. The other small thing is moving the code at "nxn" to just before the "L35" label, where the code was doing a JMP NXN, now it can just fall through. This eliminates other JMPs as well.

Here's the update "improv1" code:

```
tmp1        equ    $00
tmp2        equ    $01
number      equ    $02     ; and $03
index       equ    $04     ; and $05
array       equ    $2000

            org    $1000
sieve       ldy    #0
            lda    #$ff
l1          sta    array,y
            iny
            bne    l1

            sty    number+1      ; Start with number=2
            lda    #2
            sta    number

mbuc        lda    number
            sta    tmp1
            and    #7
            tax
            lda    number+1
            lsr                  ; y = number/8
            ror    tmp1
            lsr
```

```
            ror    tmp1
            lsr
            ror    tmp1
            ldy    tmp1

            lda    bit_expand,x  ; A = 1 << X
l3          and    array,y       ; check bit
            bne    l35           ; not prime

nxn         inc    number        ; number++
            bne    mbuc
            inc    number+1
l5          lda    number+1      ; if (number & 0x7ff) != 0 continue
            cmp    #8
            bcc    mbuc

theend      rts

l35         lda    number        ; number is prime. print it
            sta    tmp1
            lda    number+1
            sta    tmp2          ; tmp1,tmp2: data to be printed
            ldy    #0
prn1        ;--------- divide tmp1,tmp2 by 10. Remainder in A
            ldx    #16
            lda    #0
dv1         asl    tmp1
            rol    tmp2
            rol
            cmp    #10
            bcc    dv2
            sbc    #10
            inc    tmp1
dv2         dex
            bne    dv1
            ;-------------
            clc
            adc    #$b0
            pha
            iny
            lda    tmp1
            ora    tmp2
            bne    prn1
            ;-------------
prn2        pla
```

```
            jsr    cout
            dey
            bne    prn2
            lda    #$a0
            jsr    cout
            ;------------- Mark every multiple of number as not prime
            lda    number         ; index=number
            sta    index
            lda    number+1
            sta    index+1
buc2        clc                   ; index+=number
            lda    index
            adc    number
            sta    index
            sta    tmp1
            and    #7             ; X = number & 7
            tax
            lda    index+1
            adc    number+1
            sta    index+1

            cmp    #8             ; if (index >= $800) break
            bcs    nxn

            lsr                   ; y = index/8
            ror    tmp1
            lsr
            ror    tmp1
            lsr
            ror    tmp1
            ldy    tmp1

            lda    bit_exp_neg,x ; A = ~(1 << (number & 7))
            and    array,y        ; mark the bit
            sta    array,y
            jmp    buc2

cout        rts

bit_expand db      1,2,4,8,$10,$20,$40,$80
bit_exp_neg db     $fe,$fd,$fb,$f7,$ef,$df,$bf,$7f
```

This is a little shorter (182 bytes instead of 202), and now it takes just 859,661 cycles. This is now a 37% improvement.

4.3. "Improv3" Code

I then commented out the code from L35 to PRN2 to see how much time is spent preparing the decimal number for printing. This runs in 445,015 cycles, so just preparing to print the prime numbers is taking 414,646 cycles. This is worth fixing.

There are two approaches: make the binary to decimal conversion faster, which is definitely possible. Or, keep a "numberbcd" copy of number, which increments whenever "number" increments, but in BCD mode. For this program, this turns out to be faster. This is a common technique in 6502 games to track scores and other user-visible state in BCD to save on the conversion cost. This code still needs leading-0-removal logic when printing, which John Brooks optimized.

Lucas Scharenbroich shared a tip: the right shift by 3 of number+1:number at the "MBUC" label can be simplified using a small lookup to deal with the 3 bits in number+1, and then shifting just the low 8 bits right 3 times and ORA'ing the shifted upper bits. This save 10 cycles through each of 2046 loops.

This is "improv3":

```
tmp1        equ    $00
tmp2        equ    $01
number      equ    $02      ; and $03
index       equ    $04      ; and $05
numberbcd   equ    $06      ; and $07
array       equ    $2000

            org    $1000
sieve       ldy    #0
            lda    #$ff
l1          sta    array,y
            iny
            bne    l1
```

```
            sty    number+1      ; Start with number=2
            sty    numberbcd+1
            lda    #2
            sta    number
            sta    numberbcd

mbuc        lda    number+1      ; From 0…7
            tay
            lda    number
            lsr
            lsr
            lsr
            ora    shift5,y
            tay                  ; y = number/8
            lda    number
            and    #7
            tax

            lda    bit_expand,x  ; A = 1 << X
l3          and    array,y       ; check bit
            bne    l35           ; not prime

nxn         sed
            lda    numberbcd
            clc
            adc    #1
            sta    numberbcd
            bcc    nxn2
            lda    numberbcd+1
            adc    #0
            sta    numberbcd+1
nxn2        cld
            inc    number        ; number++
            bne    mbuc
            inc    number+1
l5          lda    number+1      ; if (number & 0x7ff) != 0 continue
            cmp    #8
            bcc    mbuc

theend      rts

l35
            ldy    #0            ; only 0 digits seen so far
            lda    numberbcd+1
```

```
            beq     prlowbyte
            jsr     prbyte
prlowbyte   lda     numberbcd
            jsr     prbyte

            lda     #$a0
            jsr     cout
            ;------------- Mark every multiple of number as not prime
            lda     number        ; index=number
            sta     index
            lda     number+1
            sta     index+1
buc2        clc                   ; index+=number
            lda     index
            adc     number
            sta     index
            sta     tmp1
            and     #7            ; X = number & 7
            tax
            lda     index+1
            adc     number+1
            sta     index+1

            cmp     #8            ; if (index >= $800) break
            bcs     nxn

            lsr                   ; y = index/8
            ror     tmp1
            lsr
            ror     tmp1
            lsr
            ror     tmp1
            ldy     tmp1

            lda     bit_exp_neg,x ; A = ~(1 << (number & 7))
            and     array,y       ; mark the bit
            sta     array,y
            jmp     buc2

prbyte      tax
            lsr
            lsr
            lsr
            lsr
            jsr     chkzero
```

```
            txa
            and     #$0f
chkzero     iny
            bmi     prdigit
            tay                     ; Is A non-0?
            bne     prdigit
            rts
prdigit     ora      #$b0
            tay                     ; Y > $80, print all digits after
cout        rts

shift5      db      $00,$20,$40,$60,$80,$a0,$c0,$e0
bit_expand db       1,2,4,8,$10,$20,$40,$80
bit_exp_neg db      $fe,$fd,$fb,$f7,$ef,$df,$bf,$7f
```

With this change (and PRBYTE logic to strip out leading 0's), the runtime is now 487,625 clocks and 195 bytes. This is more than twice as fast.

4.4. "Brooks1" Code

John Brooks, an expert 6502 programmer, offered some further improvements. He's investigated Sieve on the 6502 before, and so has experience in this area.

His key insight is that encoding number:number+1 differently would save a lot of shifting. In Number+1, encode the 8-bit offset into Array, and encode the bit number in the high 3 bits of Number. But: it's handy to have the bit offset in the low 3-bits of Number, so do that too! So, the initial number is (2<<5 | 2) = $42. Increment by ((1 << 5) | 1) = 33, and keep masking it by $e7 to avoid having the bits from the low 3 bits overflow into the top 3 bits.

```
* Sieve prime calc for 2^11 integers
*
* 1st 6502 version by Jesus Arias
* with mods by Kent Dickey
*
* code-golfed 11/15/2023 by JBrooks

decnum      equ     $00             ; and $01
```

```
number      equ     $02             ; and $03
index       equ     $04             ; and $05

array       equ     $2000

            org     $1000

sieve       ldx     #2
            stx     decnum
            ldx     #0
            stx     decnum+1
            ldy     #$80
            lda     #$ff
setarray    dey
            sta     array,y
            sta     array+$80,y
            bne     setarray
            lda     #2*32+2         ; y = number+1 == 0
            bne     chknum          ; always

numhi       iny                     ; y = number+1
            bne     chknum          ; if (32*number <= 0xffff) continue

exit        rts

nextnum     sed
            clc
            lda     decnum
            adc     #1
            sta     decnum
            bcc     nextnum2
            lda     decnum+1
            adc     #0
            sta     decnum+1
nextnum2    cld                     ; Carry is always clear here
            lda     number          ; number++
            adc     #1*32+1
            and     #$e7            ; lo 3 bits of number is in top 3
            bcs     numhi           ;  and lower 3 bits

chknum      sta     number
            and     #7
            tax
            lda     bitshift,x
            and     array,y
```

```
            beq     nextnum

gotprime    sty     number+1        ; number is prime. print it

prbcd       ldy     #0
            lda     decnum+1
            beq     skipzero
            jsr     prbyte
skipzero    lda     decnum
            jsr     prbyte
            lda     #$a0
            jsr     cout

clrothers   lda     number          ; index=number
            sta     index
            sta     mod1+1
            ldy     number+1
            sty     mod2+1
            clc
            bcc     clrnext         ; always

clearbit    lda     bitmask,x
            and     array,y         ; clear the bit
            sta     array,y

clrnext     lda     index           ; index+=number
mod1        adc     #0              ; self-mod #number
            and     #$e7            ; lo 3 bits of number is in both
            sta     index           ;  top & btm 3 bits
            and     #7              ; x = number & 7
            tax
            tya
mod2        adc     #0              ; self-mod #number+1
            tay

            bcc     clearbit        ; if (32*index >= $ffff) break

            ldy     number+1
            jmp     nextnum

bitshift    db      $01,$02,$04,$08,$10,$20,$40,$80
bitmask     db      $fe,$fd,$fb,$f7,$ef,$df,$bf,$7f

prbyte      tax
            lsr
```

```
            lsr
            lsr
            lsr
            jsr    chkzero
            txa
            and    #$0f
chkzero     iny
            bmi    prdigit
            tay
            bne    prdigit
            rts            ; skip leading zeroes
prdigit     ora    #$b0
            tay            ; disable zero skipping

cout        jmp    $fded
```

This code takes just 304,731 cycles, and is 174 bytes, and is more than 3 times as fast as the original code.

4.5. Further Algorithmic improvements

There is almost no end to algorithmic improvement to Sieve-like algorithms, approaching something like one line: puts(“2 3 5 7 11 …”). So it requires some sort of agreement on what is a valid optimization, or some sort of limit on size, etc.

To me, a valid sieve algorithm needs to work with various limits, at least able to work correctly for smaller lengths (like this example, it cannot easily be made longer, but it can be made shorter) as a compile-time constant.

It would be reasonable to apply the optimization to stop crossing off once Number exceeds sqrt(2048)=46. And it could be reasonable to track P*P as the position to start crossing off, to save some work. P*P can be calculated with 2 adds each time P increments. Start P=2, PXP=4 (this is P*P), PINC=5. The next P is P=P+1; PXP=PXP+PINC; and PINC=PINC+2. So maintaining PXP takes two extra adds each time P increments.

To limit the length, I also produced a 6502 version which used 310 bytes to encode the BCD difference between primes in one byte each, along with code to print it out, with a total size of 392 bytes. This runs in 48,733 clocks. To eliminate this type of "optimization", a limit on the code size allowed would be helpful, say 220 bytes.