On the 6502
Update

1 The Eratosthenes sieve again.

A lot of time passed since I wrote the original document about the quirks and performance of the 6502, but finally some people reacted to it. John Brooks and Kent Dickey pointed out the 6502 code could be made to run much faster than stated, and they were right. Following Kent tips I was able to reduce the execution time of the 6502 to about 1/3 of the original code. So, it is now a clear winner, right?

Well, not really because the same optimizations could also be applied to the Z80 code with a similar reduction in its execution time. And what about the BN16? I'm no longer maintaining that design because now I have better cores designed by myself that are already running programs in FPGAs. The last one, the BAC, is a really tiny 8-bit processor, even more spartan than the 6502, and this makes it a good candidate por the sieve contest.

Let me present first the big changes in the sieve code:

- The sieve stores only odd integers. This halves the sieve size along with saving a lot of time.
- Even prime multiples are skipped when marking not primes. (They aren't present in the new sieve, anyway)
- The binary to decimal conversion is avoided by means of keeping a copy of the number under test coded as BCD, so, no divisions by 10 are needed at all.
- The 8 possible values of the bit mask are stored in a table instead of being computed. The sieve bits are inverted (0 means prime) in order to use the same table for bit testing and bit marking.

And also some other processor specific optimizations were included, like using more the zero page in the 6502 or the RLD instruction in the Z80 case for 4-bit shifts during number printing.

And here are the new results:

	6502	Z80	Z80/6502	BAC	BAC/6502
Code size	259×8	232×8	0.90	157×16	1.23
Clock cycles	407708	775184	1.90	129856	0.32
Exec. Instructions	129024	107208	0.83	123245	0.95
Clock frequency (MHz)	1	2.5	2.5	48	48
Exec. time (ms)	408	310	0.76	2.7	0.0066
Average Cycles/Instr.	3.16	7.23	2.29	1.05	0.33

Here, I'm using the lowest clock frequency rating for the old CPUs instead of the average values of the previous document when comparing times. The idea is to compare the capabilities of the CPUs instead of systems.

Just for comparison, let me present the old numbers (before optimizations):

	6502	Z80	Z80/6502
Code size	202×8	144×8	0.71
Clock cycles	1162093	2282458	1.96
Exec. Instructions	384373	304526	0.79
Clock frequency (MHz)	1.305	3.571	2.74
Exec. time (ms)	890	639	0.718
Average Cycles/Instr.	3.02	7.49	2.48

In summary, the improved codes run much faster than before but the ratios between processors show only minor changes. In any case, the clear winner is the little BAC uC, that, BTW, has no support for BCD arithmetic at all.

But one thing are my codes an other completely different those of John Brooks. He devised an algorithm that computed the primes in just 75812 cycles (17165 cycles for the sieve, the rest for printing) in a 6502! And he was also patient enough to port its sieve to the Z80, resulting in 32847 cycles for the sieve. Both results are impressive, but you know what? 32847/17156 is 1.91, the same clock ratio we get every time we compare a Z80 code versus its equivalent 6502 code.

2 Annexes

2.1 The BAC computer

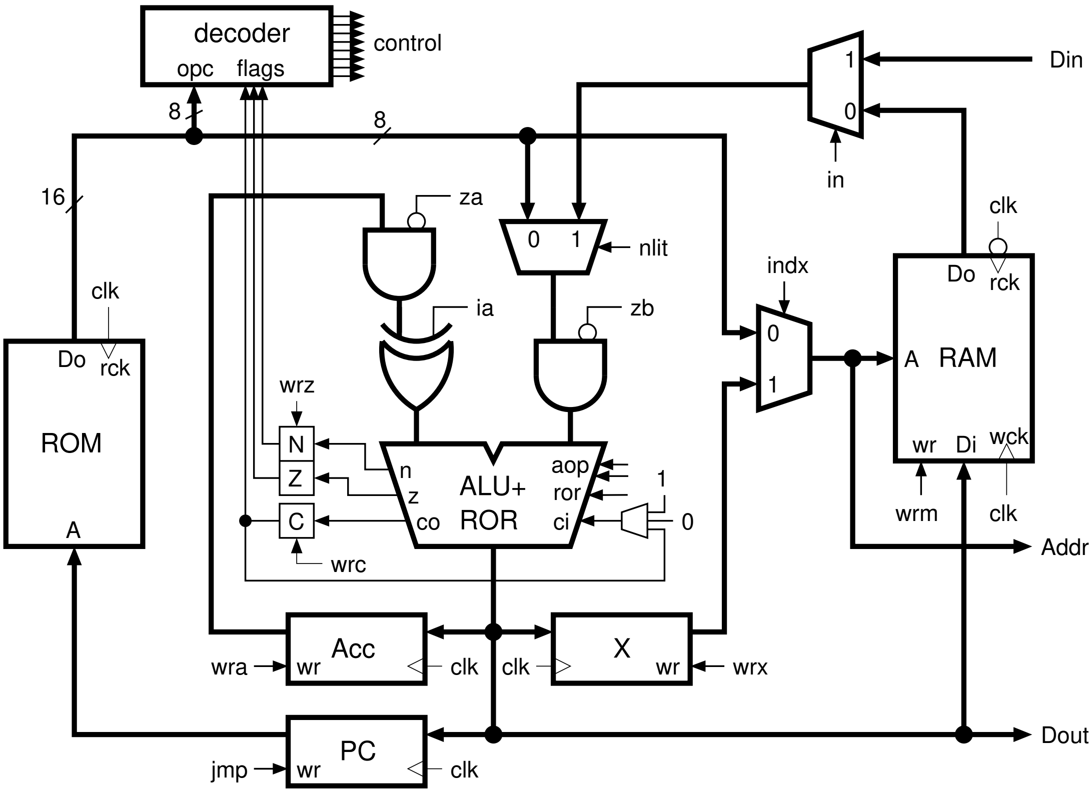

Figure 1: Block diagram of the BAC processor

This is a very small 8-bit computer designed as a microcontroller core for FPGAs. The sieve test was also ported to this processor, and as we can see in the above table, it was the winner for almost every comparison except code size. It is worth mentioning this core was fit into only 156 logic cells in the FPGA, while a 6502 replica required 673 logic cells and a Z80 replica no less than 2247 logic cells. And, yes, with only 8-bit address buses it is quite limited (It is comparable to the PIC10F200 microcontroller).

In the BAC processor we got only two registers, Acc and X, appart from the program counter and flags. X is a write only register used as a memory pointer. There is no stack. The main idea is to store all the needed variables in RAM and to use the registers as a temporary place for RAM data when needed.

In the BAC processor instructions are 16-bit wide, and they follow this format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Instruction						INDX	NLIT	Literal data							

The instructions are:

Instruction	Flags	Mnemonic	Description
0000.0x	-,-,-	NOP	
0000.10	-,-,-	JMP	unconditional jump
0000.11	-,-,-	JMPD	unconditional jump, delayed
0001.00	-,-,-	JNC	jump if C==0
0001.01	-,-,-	JNCD	jump if C==0, delayed
0001.10	-,-,-	JC	jump if C==1
0001.11	-,-,-	JCD	jump if C==1, delayed
0010.00	-,-,-	JNZ	jump if Z==0
0010.01	-,-,-	JNZD	jump if Z==0, delayed
0010.10	-,-,-	JZ	jump if Z==1
0010.11	-,-,-	JZD	jump if Z==1, delayed
0011.00	-,-,-	JPL	jump if N==0
0011.01	-,-,-	JPLD	jump if N==0, delayed
0011.10	-,-,-	JMI	jump if N==1
0011.11	-,-,-	JMID	jump if N==1, delayed
0100.00	N,Z,-	LDA	Acc = op
0100.01	N,Z,-	IN	Acc = IO[addr]
0100.10	-,-,-	LDX	X = op
0101.00	-,-,-	STA	Mem[addr] = Acc
0101.01	-,-,-	OUT	IO[addr] = Acc
0101.10	-,-,-	TAX	X = Acc
1000.00	N,Z,C	ADDA	Acc = op + Acc
1000.01	N,Z,C	ADDM	Mem[addr] = Mem[addr] + Acc
1000.10	N,Z,C	ADCA	Acc = op + Acc + C
1000.11	N,Z,C	ADCM	Mem[addr] = Mem[addr] + Acc + C
1001.00	N,Z,C	SUBA	Acc = op - Acc
1001.01	N,Z,C	SUBM	Mem[addr] = Mem[addr] - Acc
1001.10	N,Z,C	SBCA	Acc = op - Acc - /C
1001.11	N,Z,C	SBCM	Mem[addr] = Mem[addr] - Acc - /C
1010.00	N,Z,C	CMP	op - Acc (result not stored)
1010.01	N,Z,-	TST	op & Acc (result not stored)
1010.10	N,Z,C	ROR	Mem[addr] = $\{C, Mem_{7:1}\}$, C=Mem_0
1011.00	N,Z,-	ANDA	Acc = op & Acc
1011.01	N,Z,-	ANDM	Mem[addr] = Mem[addr] & Acc
1011.10	N,Z,-	ORA	Acc = op | Acc
1011.11	N,Z,-	ORM	Mem[addr] = Mem[addr] | Acc
1100.00	N,Z,-	XORA	Acc = op ^ Acc
1100.01	N,Z,-	XORM	Mem[addr] = Mem[addr] ^ Acc
1101.00	N,Z,-	INC	Mem[addr] = Mem[addr] + 1
1101.01	N,Z,-	INCA	Acc = Mem[addr] = Mem[addr] + 1
1101.10	N,Z,-	INCX	X = Mem[addr] = Mem[addr] + 1
1101.11	N,Z,-	INCAX	Acc = X = Mem[addr] = Mem[addr] + 1
1110.00	N,Z,-	DEC	Mem[addr] = Mem[addr] - 1
1110.01	N,Z,-	DECA	Acc = Mem[addr] = Mem[addr] - 1
1110.10	N,Z,-	DECX	X = Mem[addr] = Mem[addr] - 1
1110.11	N,Z,-	DECAX	Acc = X = Mem[addr] = Mem[addr] - 1

Bits 8 and 9 select the addressing mode for the instruction:

INDX,NLIT	Addressing mode
x0	Literal
01	Direct
11	Indexed

Beware of the order of the operands in the subtraction and comparison instructions: **Acc is the value that is subtracted** (this order is similar to that of the W register in 8-bit PIC microcontrollers).

In the instruction set we can see there is a delayed version of each jump instruction. That's because the ROM memory is synchronous and that means all instructions are executed with one cycle delay. This only poses a problem for jump instructions because when you jump, the instruction that was after the jump has already been loaded into the ROM output register, and if nothing is done about it it will be executed.

This has been prevented by having the current instruction executed as a NOP if the previous instruction was a taken jump. In this way, the unconditional jump lasts two effective clock cycles to execute, and conditional jumps lasts two cycles when taken or only one if not taken.

But the delayed jump behavior have also been preserved. In these jumps, the instruction that follows them is not invalidated and therefore it will always be executed, also for the conditional jumps. It is as if the jump were executed one instruction later than it should be according to its position in the program, and that's why we call it delayed. We can take advantage of this behavior by placing useful instructions after the jumps. We can also exploit these jumps for addressing tables of constants from the program memory, as is shown in this example:

```
; Print string from program memory
; par1 : pointer to the beginning of the ASCIIZ string
; trick learned from the GIGATRON computer
pputs:  decx    [sp]   ; save return address
        sta     [x]
pp1:    jmpd    [par1] ; Jump to table and run LDA n
        jmpd    .+1    ; But immediately jump back (to JZD)
        jzd     pp2
        inc     [par1] ; pointer++
        jmpd    pp1
        out     [0]    ; data to terminal
pp2:    ldx     [sp]   ; subroutine return
        jmpd    [x]
        inc     [sp]
; text strings stored in program memory
txt:    lda     'H'
        lda     'e'
        lda     'l'
        lda     'l'
        lda     'o'
        lda      0
```

2.2 6502 codes

2.2.1 Author' s sieve

```
;------------------------- 6502 Erat. Sieve, improved ------------------------
;---    J. Arias (2023)
        *=0                             ; Zero Page Vars
        tmp1:           *=*+1
        tmp2:           *=*+1
        number:         *=*+2
        index:          *=*+2
        nbcd:           *=*+2
        array:          *=*+2           ; 128 byte array
        *=$e000
direxe: ldx     #127            ; Mark all numbers as primes to begin
        lda     #$00
l1:     sta     array,x
        dex
        bpl     l1
        lda     #'1'            ; 1 is prime, print
        jsr     cout
        lda     #' '
        jsr     cout
        lda     #'2'            ; 2 is prime, print
        jsr     cout
        lda     #' '
        jsr     cout
        lda     #0
        sta     number+1        ; start with number=3
        sta     nbcd+1
        lda     #3
        sta     number
        sta     nbcd
mbuc:   lda     number          ; check if prime
        sta     tmp1
        lda     number+1
        sta     tmp2
        lsr     tmp2            ; y = number/16
        ror     tmp1
        lda     tmp1            ; A = (number>>1)
        lsr     tmp2
        ror     tmp1
        lsr     tmp2
        ror     tmp1
        lsr     tmp2
        ror     tmp1
        ldx     tmp1
        and     #7              ; A = 1<<((number>>1)&7)
        tay
        lda     tab,y
        and     array,x         ; check bit
        beq     l35             ; not prime
        jmp     nxn
        ; number is prime. print it
l35:    lda     nbcd+1
        beq     l37
        ror
        ror
        ror
        ror
        and     #$0f
```

```
        beq     l36
        clc
        adc     #48
        jsr     cout
l36:    lda     nbcd+1
        beq     l37
        and     #$0f
        clc
        adc     #48
        jsr     cout
l37:    lda     nbcd
        ror
        ror
        ror
        ror
        and     #$0f
        bne     l371
        ldx     nbcd+1
        beq     l38
l371:   clc
        adc     #48
        jsr     cout
l38:    lda     nbcd
        and     #$0f
        clc
        adc     #48
        jsr     cout
        lda     #32
        jsr     cout
        ;--------------- Now, 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     #1              ; skip even indexes
        and     index
        beq     buc2
        lda     #7              ; if (index>=$800) break
        cmp     index+1
        bcc     nxn
        lsr     tmp2            ; y = index/16
        ror     tmp1
        lda     tmp1            ; A = (index>>1)
        lsr     tmp2
        ror     tmp1
        lsr     tmp2
        ror     tmp1
        lsr     tmp2
        ror     tmp1
        ldx     tmp1

        and     #7              ; A = ~(1<<((index>>1)&7))
        tay
```

```
        lda     tab,y
l7:     ora     array,x         ; mark the bit
        sta     array,x
        jmp     buc2
nxn:    ; next prime number
        clc                     ; number+=2
        lda     number
        adc     #2
        sta     number
        lda     number+1
        adc     #0
        sta     number+1
        lda     number+1        ; if (number&0x7ff)!=0 continue
        cmp     #8
        beq     theend
        ; update BCD
        sed
        clc
        lda     nbcd
        adc     #2
        sta     nbcd
        lda     nbcd+1
        adc     #0
        sta     nbcd+1
        cld

        jmp     mbuc
theend:
        brk             ; stop simulation
        jmp theend
cout:                   ; character output routine
        ;sta    $240    ; emulated <stdout> comment for speed test
        rts
tab:    .byte 1,2,4,8,16,32,64,128
```

2.2.2 John Brook's fast sieve (for Apple II)

```
*-------------------------------
* Sieve of Eratosthenes for Apple II
* Merlin-16 v3.5.1 assembler
* 11/25/2023 by John Brooks
*-------------------------------
* Primes less than 2048 in 276 code bytes
* 17,165 cycle prime calc (8110:4C to time)
* 75,812 cycle prime w/print (8177:60 to time)
*-------------------------------
* Primes less than 61,440 in 288 code bytes
*   913,120 cycle prime calc (8110:4C to time)
* 2,118,067 cycle prime w/print (8183:60 to time)
*-------------------------------
               lst   on          ; Merlin assembler: generate assembly listing
               org   $8100       ; Code execution starts at $8100
Sieve2048      equ   1           ; 0 = 66K primes, 1 = 2K primes
ZpCodeOrg      equ   $3a         ; zero page address where Prime calc code runs
*...........
               do    Sieve2048
PrimeRange     equ   2048        ; check integers less than 2048 for primes
OddRange       equ   PrimeRange/2 ; check 1024 odd integers
Flags          equ   $8000-OddRange ; 1024 byte-per-odd flags array ends at $8000
BcdTmp         equ   $00         ; holds BCD low digit while printing high digit
NumAsBcd       equ   $01         ; 4-digit BCD of odd-integers during print
SpaceChar      equ   " "         ; display a space between primes
*...........
               else
PrimeRange     equ   $f000       ; check integers less than 66,140 for primes
OddRange       equ   PrimeRange/2 ; check $7800 odd integers
Flags          equ   $8000-OddRange ; byte-per-odd flags array at $800-$8000
BcdTmp         equ   $00         ; holds BCD low digit while printing high digit
NumAsBcd       equ   $01         ; 5-digit BCD of odd-integers during print
SpaceChar      equ   $8D         ; display one prime per line
               fin
*...........
               mx    %11         ; Merlin: 8-bit mem/acc, 8-bit xy regs
Sieve
               ldx   #SieveEnd-SieveZP+1 ; Num bytes to copy to ZP. +1 for X=0 exit
CopyToZp
               lda   RelocToZP-1,x
               sta   ZpCodeOrg-1,x
               dex
               bne   CopyToZp
               ldy   #0          ; y: Flags index == 0
               ldx   #15-1       ; x: wheel constants index
               lda   #%01110110 ; initial primes = .,3,5,7,.,11,13,.
               jsr   SetFlag1    ; find all primes
*-------------------------------
DispPrimes
               lda   #"2"        ; display the single even prime: 2
               jsr   Cout
               lda   #SpaceChar
               jsr   Cout
               ldy   #$00
               lda   #$03        ; first BCD number checked is == $0003
*...........
               do    Sieve2048
               sty   NumAsBcd+1
*...........
               else
```

```
               sty   NumAsBcd+1
               sty   NumAsBcd+2
               fin
*...........
               iny              ; y: start checking number 3, Flags index (3/2 == 1)
               sed              ; enable 6502 BCD mode
               clc              ; c=0 assumed in loop
               bne   DispChk    ; always
DispNextBcdH
               tax              ; save BcdL
               lda   NumAsBcd+1 ; BcdH++
               adc   #0
               sta   NumAsBcd+1
*...........
               do    Sieve2048
               txa              ; restore BcdL
*...........
               else
               bcc   DispBcd5Ok
               inc   NumAsBcd+2
               clc
DispBcd5Ok
               txa              ; restore BcdL
               fin
*...........
               iny              ; check PtrL++
               bne   DispChk
DispNextPtrH
               inc   DispChk+2  ; check PtrH++
               bpl   DispChk
DispExit
               cld              ; disable BCD mode
               rts
DispBcd
               cld              ; disable BCD mode during Cout print
*...........
               do    Sieve2048
               ldx   NumAsBcd+1 ; set X<$80 to skip printing leading zero digits
               beq   DispSkip00
               txa              ; non-zero in top two BCD digits, print them
*...........
               else
               ldx   NumAsBcd+2 ; set X<$80 to skip printing leading zero digits
               beq   DispNoBcd5
               txa
               jsr   DispDigit
DispNoBcd5
               lda   NumAsBcd+1
               fin
*...........
               jsr   DispByte
DispSkip00
               lda   NumAsBcd   ; print low two BCD digits
               jsr   DispByte
               lda   #SpaceChar ; print space
               jsr   Cout
               lda   NumAsBcd   ; acc: BcdL
               sed              ; 6502 BCD mode enabled
               clc              ; loop assumes c=0
DispNext
               adc   #2         ; check BcdL += 2
               bcs   DispNextBcdH
```

```
                iny                 ; check PtrL++
                beq   DispNextPtrH
DispChk         ldx   Flags,y       ; self-mod PtrH
                bpl   DispNext      ; branch if not prime
                sta   NumAsBcd      ; save acc:BCD for printing
                bmi   DispBcd       ; always
*-------------------------------
DispZero
                inx                 ; x: > 128 if a non-zero digit has printed
                bmi   DispDigit
                rts                 ; skip leading zeroes
*-------------------------------
DispByte
                sta   BcdTmp        ; save BCD low digit
                lsr                 ; shift BCD high to low
                lsr
                lsr
                lsr
                jsr   ChkZero
                lda   BcdTmp        ; get low digit
                and   #$0f
ChkZero
                beq   DispZero
DispDigit
                ora   #"0"          ; make ascii 0-9
                tax                 ; disable zero skipping for the rest of the number
Cout            jmp   $fded         ; Apple II ROM character output routine
*-------------------------------
RelocToZP
                org   ZpCodeOrg
SieveZP
* wheel of primes for odd integers less than 2*3*5*7 (210 integers)
* stored as 15 * 7 bits (105 bits for the odd integers < 2*3*5*7)
Wheel210
                db    %11000010  ; 197,199,...,...,...,...,209,0
                db    %00101100  ; ...,...,187,...,191,193,...,0
                db    %10100110  ; 169,...,173,...,...,179,181,0
                db    %01001010  ; ...,157,...,...,163,...,167,0
                db    %01001100  ; ...,143,...,...,149,151,...,0
                db    %10100110  ; 127,...,131,...,...,137,139,0
                db    %10001000  ; 113,...,...,...,121,...,...,0
                db    %01101100  ; ...,101,103,...,107,109,...,0
                db    %00100010  ; ...,..., 89,...,...,..., 97,0
                db    %11001010  ;  71, 73,...,..., 79,..., 83,0
                db    %01100100  ; ..., 59, 61,...,..., 67,...,0
                db    %10100100  ;  43,..., 47,...,..., 53,...,0
                db    %11001010  ;  29, 31,...,..., 37,..., 41,0
                db    %01101000  ; ..., 17, 19,..., 23,...,...,0
                db    %10000110  ;   1,...,...,...,..., 11, 13,0
*-------------------------------
SetFlagPtrH                         ; self-mod writes to the Flags array (PtrH)
                sta   SetFlag1+2
                sta   SetFlag2+2
                sta   SetFlag3+2
                sta   SetFlag4+2
                sta   SetFlag5+2
                sta   SetFlag6+2
                sta   SetFlag7+2
SetFlagPtrL                         ; self-mod writes to the Flags array (PtrL)
                stx   SetFlag1+1
                inx
                stx   SetFlag2+1
```

```
                inx
                stx   SetFlag3+1
                inx
                stx   SetFlag4+1
                inx
                stx   SetFlag5+1
                inx
                stx   SetFlag6+1
                inx
                stx   SetFlag7+1
*-------------------------------
DoWheel210
                ldx   #15-1      ; load 15 7-bit wheel constants
DoWheelByte
                lda   Wheel210,x ; acc: 7 bits of wheel constants
                                 ; set bit 7 of Flags: 1=check for prime, 0=not prime
SetFlag1        sta   Flags,y
                asl
SetFlag2        sta   Flags+1,y
                asl
SetFlag3        sta   Flags+2,y
                asl
SetFlag4        sta   Flags+3,y
                asl
SetFlag5        sta   Flags+4,y
                asl
SetFlag6        sta   Flags+5,y
                asl
SetFlag7        sta   Flags+6,y
                tya              ; y: Flags index += 7
                clc
                adc   #7
                tay
                dex              ; x: next wheel constant
                bpl   DoWheelByte ; loop for 15 wheel bytes
                asl
                bcc   DoWheel210 ; loop while y:FlagsIndex < 128
                lsr              ; y: Flags index &= $7F to avoid y overflow
                tay
                lda   SetFlag1+1 ; Flags PtrL += $80 to avoid y overflow
                eor   #$80
                tax
                bmi   SetFlagPtrL ; Update 7 Flags PtrL
                lda   SetFlag1+2 ; PtrH++
                adc   #1         ; C=0 from lsr above
                bpl   SetFlagPtrH ; Update 7 Flags PtrL & PtrH
*-------------------------------
* Check Flags starting at number 11. Wheel210 has excluded multiples of 3,5,7
* Start excluding Flags at Prime squared, 11*11
                ldy   #11        ; y: Prime check = 11
                ldx   #>11*11/2+Flags ; xa: Prime squared = 11^2. Div2 for only-odd
                lda   #<11*11/2+Flags
ChkPrime
                sta   ModSq+1    ; save acc
                inc   ModChkPtr+1 ; ++FlagsPtr
ModChkPtr       bit   11/2+Flags-1 ;acc: OddPrimeFlag
                bpl   ChkNext    ; branch if not prime, ie < 128
*-------------------------------
* Exclude multiples of the found prime
* Start at prime^2 (ptr in xa)
                stx   ModExcOk+1 ; save x:Flags PtrH
                sty   ModExcInc+1 ; set stride to exclude Flags
```

```
SetExcPtrH
               stx   ModExcPtr+2 ; set PtrH
               clc               ; assumes c=0 in loop
ExcLoop
               sta   ModExcPtr+1 ; set PtrL
ModExcPtr      sty   $ff00       ; exclude Flags entry via bit7=0 (y always < 128)
ModExcInc      adc   #0          ; step to next Flags entry to exclude
               bcc   ExcLoop     ; exclude all flags in the page
               inx               ; PtrH++
               bpl   SetExcPtrH  ; set PtrH
ModExcOk       ldx   #0          ; restore x: prime^2 PtrH
*-------------------------------
ChkNext
               iny               ; y: prime chk += 1
               tya
               iny               ; y: prime chk += 1
               asl               ; incrementally update prime^2
ModSq          adc   #0          ; add to old xa:prime^2 ptr
               bcc   ChkPrime
               inx               ; PtrH++
               bpl   ChkPrime
               rts               ; All primes found when prime^2 ptr >= $8000
SieveEnd
               lst   off         ; Merlin: disable listing the entire symbol table
```

2.3 Z80 codes

2.3.1 Author's sieve

```
;-------------------------- Z80 Erat. Sieve, improved ------------------------
;--- J. Arias (2023)
        MAXP equ 2048
        ; DE: number
        ; HL: index

        org    0x0
        ld      a,'1'
        call    cout
        ld      a,' '
        call    cout
        ld      a,'2'
        call    cout
        ld      a,' '
        call    cout
        ld      hl,array        ;Mark all numbers as primes to begin
        ld      de,array+1
        ld      bc,MAXP/16-1
        ld      a,0
        ld      (hl),a
        ldir

        ld      de,3            ; start with number=3
        ld      (nbcd),de

mbuc:   ld      h,d
        ld      l,e
        srl     h               ; l=number/16
        rr      l
        ld      a,l
        srl     h
        rr      l
        srl     h
        rr      l
        srl     h
        rr      l
        ld      bc,array
        add     hl,bc
        ; A = 1<<((number>>1)&7)
        and     7
        exx
        ld      hl,tab
        add     a,l
        ld      l,a
        ld      a,(hl)
        exx
        and     (hl)
        jr      nz,nxp
        ;------------- number is prime ---------
        ;----- print it -----
        ld      bc,(nbcd)   ; save a copy in cpbcd for RLD
        ld      (cpbcd),bc
        ld      hl,cpbcd+1  ; point to Thousands, hundreds
        xor     a
        rld
        jr      z,pr1
        add     a,48
```

```
        call    cout
pr1:    ld      a,b
        or      a
        jr      z,pr2
        xor     a
        rld
        add     a,48
        call    cout
pr2:    dec     hl      ; Point to tens, units
        xor     a
        rld
        jr      nz,pr3
        or      b
        jr      z,pr4
        xor     a
pr3:    add     a,48
        call    cout
pr4:    xor     a
        rld
        add     a,48
        call    cout

        ld      a,32    ; space
        call    cout

        ;--------------- Now, mark every multiple of number as not prime
        ld      h,d
        ld      l,e
buc2:   add     hl,de

        ;------ if (HL>=$1000) break
        ld      a,h
        cp      MAXP/256
        jr      nc,nxp
        ;------ if (HL even) continue
        bit     0,l
        jr      z,buc2

        push    hl
        ld      a,l
        srl     h           ; l=index/16
        rr      l
        ld      a,l
        srl     h
        rr      l
        srl     h
        rr      l
        srl     h
        rr      l
        ld      bc,array
        add     hl,bc
        and     7
        exx
        ld      hl,tab
        add     a,l
        ld      l,a
        ld      a,(hl)
        exx
        or      (hl)
        ld      (hl),a    ; mark the bit
        pop     hl
        jr      buc2
```

```
        ;------------ not prime -----------
nxp:    inc     de          ; number+=2
        inc     de
        ld      a,d
        cp      MAXP/256
        jr      z,stop      ; if (number<MAX) continue
        ld      hl,nbcd     ; BCD copy += 2
        ld      a,2
        add     a,(hl)
        daa
        ld      (hl),a
        inc     hl
        ld      a,0
        adc     a,(hl)
        daa
        ld      (hl),a
        jp      mbuc

stop:   halt

cout:   out     (0),a        ; emulated <stdout> comment for speed test
        ret
nbcd:   db      0,0           ; BCD value
cpbcd:  db      0,0           ; space for BCD printing (for RLD)
        ds      (8-($ & 7)) & 7  ; alignement to multiple of 8
tab:    db      1,2,4,8,16,32,64,128

array:
```

2.3.2 John Brook’s fast sieve for Z80

```
;------------------------------
; Sieve of Eratosthenes for Z-80
; z80asm v1.8 assembler, tab size 8
; 11/29/2023 by John Brooks
;------------------------------
; Primes less than 2048 in xxx code bytes
; 31,728 cycle prime calc w/102 bytes
;  vs 6502 @ 17,165 cycles w/174 bytes (Z80 = 1.85x more cycles)
; xxx cycle prime w/print
;------------------------------

                org     $8100           ; Code execution starts at $8100

Sieve2048:      equ     1               ; 0 = 66K primes, 1 = 2K primes

;...........
                if      Sieve2048
PrimeRange:     equ     2048            ; check integers less than 2048 for primes
OddRange:       equ     PrimeRange/2    ; check 1024 odd integers
Flags:          equ     $8000-OddRange  ; 1024 byte-per-odd flags array ends at $8000

BcdTmp:         equ     $00             ; holds BCD low digit while printing high digit
NumAsBcd:       equ     $01             ; 4-digit BCD of odd-integers during print

SpaceChar:      equ     " "             ; display a space between primes
;...........
                else
PrimeRange:     equ     $f000           ; check integers less than 66,140 for primes
OddRange:       equ     PrimeRange/2    ; check $7800 odd integers
Flags:          equ     $8000-OddRange  ; byte-per-odd flags array at $800-$8000

BcdTmp:         equ     $00             ; holds BCD low digit while printing high digit
NumAsBcd:       equ     $01             ; 5-digit BCD of odd-integers during print

SpaceChar:      equ     $8D             ; display one prime per line
                endif
;...........

Sieve:

                ld      hl, Flags       ; 10c hl: Flags out ptr
                ld      de, Wheel210+14 ; 10c de: Wheel constants ptr
                ld      b, 15           ;  7c b: bytes in wheel table
                ld      a, %01110110    ;  7c initial primes = .,3,5,7,.,11,13,.
                jp      SetFlag1        ; 10c find all primes


; wheel of primes for odd integers less than 2*3*5*7 (210 integers)
; stored as 15 * 7 bits (105 bits for the odd integers < 2*3*5*7)
Wheel210:
                db      %11000010       ; 197,199,...,...,...,...,209,0
                db      %00101100       ; ...,...,187,...,191,193,...,0
                db      %10100110       ; 169,...,173,...,...,179,181,0
                db      %01001010       ; ...,157,...,...,163,...,167,0
                db      %01001100       ; ...,143,...,...,149,151,...,0
                db      %10100110       ; 127,...,131,...,...,137,139,0
                db      %10001000       ; 113,...,...,...,121,...,...,0
                db      %01101100       ; ...,101,103,...,107,109,...,0
                db      %00100010       ; ...,..., 89,...,...,..., 97,0
                db      %11001010       ;  71, 73,...,..., 79,..., 83,0
                db      %01100100       ; ..., 59, 61,...,..., 67,...,0
                db      %10100100       ;  43,..., 47,...,..., 53,...,0
                db      %11001010       ;  29, 31,...,..., 37,..., 41,0
                db      %01101000       ; ..., 17, 19,..., 23,...,...,0
                db      %10000110       ;   1,...,...,...,..., 11, 13,0
```

```
DoWheel210:
                ld      de, Wheel210+14 ; 10c de: Wheel constants ptr
                ld      b, 15           ;  7c: b: load 15 7-bit wheel constants
DoWheelByte:
                ld      a,(de)          ;  7c a: 7 bits of wheel constants
SetFlag1:       ld      (hl), a         ;  7c
                inc     hl              ;  6c
                add     a               ;  4c
SetFlag2:       ld      (hl), a         ;  7c
                inc     hl              ;  6c
                add     a               ;  4c
SetFlag3:       ld      (hl), a         ;  7c
                inc     hl              ;  6c
                add     a               ;  4c
SetFlag4:       ld      (hl), a         ;  7c
                inc     hl              ;  6c
                add     a               ;  4c
SetFlag5:       ld      (hl), a         ;  7c
                inc     hl              ;  6c
                add     a               ;  4c
SetFlag6:       ld      (hl), a         ;  7c
                inc     hl              ;  6c
                add     a               ;  4c
SetFlag7:       ld      (hl), a         ;  7c
                inc     hl              ;  6c

                dec     e               ;  4c (de): next wheel constant
                djnz    DoWheelByte     ; 13c loop for 15 wheel bytes

                bit     7, h            ;  8c
                jp      z, DoWheel210   ; 10c

;-------------------------------
; Check Flags starting at number 11. Wheel210 has excluded multiples of 3,5,7
; Start excluding Flags at Prime squared, 11*11
                ld      c, 11           ;  7c Prime check = 11
                ld      de, 11/2+Flags  ; 10c de: Flags ptr
                ld      hl, 11*11/2+Flags ; 10c hl: Prime squared = 11^2. Div2 for only-odd

ChkPrime:       ld      a, (de)         ;  7c check the next odd number
                inc     de              ;  6c hl: checkPtr++
                add     a               ;  4c a: a<<1 to check if prime (a >= $80)
                jr      nc, ChkNext     ;  7c/12c branch if not prime

;-------------------------------
; Exclude multiples of the found prime
; Start at de:prime^2

                push    hl              ; 11c save Prime^2
                ld      a,l             ;  4c a: exclude ptrL
ExcLoop:
                ld      (hl), h         ;  7c exclude Flags entry via bit7=0 (h always < 128)
                add     c               ;  4c step to next Flags entry to exclude
                ld      l,a             ;  4c set PtrL
                jp      nc, ExcLoop     ; 10c exclude all flags in the page
                inc     h               ;  4c PtrH++
                jp      p, ExcLoop      ; 10c Exclude until $8000

                pop     hl              ; 10c restore check ptr

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

ChkNext:
                inc     c               ;  4c prime check += 1
                ld      a, c            ;  4c
                inc     c               ;  4c prime check += 1 to next odd number
```

```
add     a               ;  4c incrementally update prime^2
add     l               ;  4c add to old de:prime^2 ptr
ld      l,a             ;  4c hl: next prime^2 ptr
jp      nc, ChkPrime    ; 10c
inc     h               ;  4c PtrH++
jp      p, ChkPrime     ; 10c Check flags until prime^2 ptr >= $8000
ret
```

2.4 BAC code

```
;------------------------- BAC Erat. Sieve, improved ------------------------
;--- J. Arias (2023)
sp=     0xff    ; Stack pointer
lr=     0xfe    ; Link register (return address for leaf subroutines)
tmp16=  0xf0
number= 0xf2
index=  0xf4
var1=   0xf6
deci0=  0xf7    ; decimal version of 'number'
deci1=  0xf8
deci2=  0xf9
deci3=  0xfa
; Peripheral registers
UDAT=   0x01
UFLAGS= 0x02
; Sieve size (up to 0xF0)
size=   128
szcmp=  (size-1)>>4
init:   ;lda        0xff          ; init SP
        ;sta        [sp]

        lda     size            ; fill sieve with zeroes
        sta     [var1]
        lda     0
l01:    decx    [var1]
        jnzd    l01
        sta     [x]

        lda     '1'             ; 1 and 2 are primes: print them directly
        sta     [var1]
        lda     .+2
        jmp     cout
        lda     ' '
        sta     [var1]
        lda     .+2
        jmp     cout
        lda     '2'
        sta     [var1]
        lda     .+2
        jmp     cout
        lda     ' '
        sta     [var1]
        lda     .+2
        jmp     cout
        lda     0               ; start with number=3
        sta     [number+1]
        sta     [deci3]         ; also init the decimal version of number
        sta     [deci2]
        sta     [deci1]
        lda     3
        sta     [number]
        sta     [deci0]
mbuc:   lda     [number]        ; check bit
        sta     [tmp16]
        lda     [number+1]
        sta     [tmp16+1]
        ror     [tmp16+1]       ; tmp16 >>= 4
        ror     [tmp16]
        lda     [tmp16]         ; var1 = (tmp16 >> 1)
```

```
        sta     [var1]
        ror     [tmp16+1]
        ror     [tmp16]
        ror     [tmp16+1]
        ror     [tmp16]
        ror     [tmp16+1]
        ror     [tmp16]
        lda     0x7
        andm    [var1]          ; var1 = (tmp16 >> 1)&7
        lda     bittab
        addm    [var1]
        jmpd    [var1]          ; program memory table read
        jmpd    .+1
        ldx     [tmp16]
        tst     [x]             ; bit tested (0 means prime)
        jnz     nxp
        ;------------- number is prime ---------
        ;----- print it -----
        lda     [deci3]         ; Thousands
        jz      pr2
        adda    '0'
        sta     [var1]
        lda     .+2
        jmp     cout
pr2:    lda     [deci2]         ; Hundreds
        ora     [deci3]
        jz      pr3
        lda     [deci2]
        adda    '0'
        sta     [var1]
        lda     .+2
        jmp     cout
pr3:    lda     [deci1]         ; Tens
        ora     [deci2]
        ora     [deci3]
        jz      pr4
        lda     [deci1]
        adda    '0'
        sta     [var1]
        lda     .+2
        jmp     cout
pr4:    lda     [deci0]         ; Units
        adda    '0'
        sta     [var1]
        lda     .+2
        jmp     cout

        lda     ' '             ; space
        sta     [var1]
        lda     .+2
        jmp     cout

;--------------- Now, mark every multiple of number as not prime
        lda     [number]
        sta     [index]
        lda     [number+1]
        sta     [index+1]
l30:    lda     [number]
        addm    [index]
        lda     [number+1]
        adcm    [index+1]
        lda     [index+1]    ; stop if outside table
```

```
        cmp     szcmp
        jnc     nxp
        lda     1               ; skip even indexes
        tst     [index]
        jz      l30
        lda     [index]
        sta     [tmp16]
        lda     [index+1]
        sta     [tmp16+1]
        ror     [tmp16+1]       ; tmp16 >>= 4
        ror     [tmp16]
        lda     [tmp16]
        sta     [var1]          ; var1 = (tmp16 >> 1)
        ror     [tmp16+1]
        ror     [tmp16]
        ror     [tmp16+1]
        ror     [tmp16]
        ror     [tmp16+1]
        ror     [tmp16]
        lda     0x7
        andm    [var1]          ; var1 = (tmp16 >> 1)&7
        lda     bittab
        addm    [var1]
        jmpd    [var1]          ; program memory table read
        jmpd    .+1
        ldx     [tmp16]
        jmpd    l30
        orm     [x]

nxp:    lda     2               ; next prime= number+2
        addm    [number]
        lda     0
        adcm    [number+1]
        lda     szcmp+1
        cmp     [number+1]
        jpl     stop
        lda     2               ; decimal copy +=2
        addm    [deci0]
        lda     10
        cmp     [deci0]
        jmi     mbuc
        subm    [deci0]
        inc     [deci1]
        cmp     [deci1]
        jmi     mbuc
        subm    [deci1]
        inc     [deci2]
        cmp     [deci2]
        jmi     mbuc
        subm    [deci2]
        jmpd    mbuc
        inc     [deci3]
stop:   brk                     ; Stop simulation (brk = NOP on real hardware)
        jmp     .
bittab: lda     1               ; table with powers of 2 in program memory
        lda     2
        lda     4
        lda     8
        lda     16
        lda     32
        lda     64
        lda     128
```

```
; print character from [var1], "leaf" routine (no stack)
cout:   sta     [lr]
;co1:    in      [UFLAGS]        ; wait for TX_rdy (sign bit)
;        jpl     co1
;        lda     [var1]
;        jmpd    [lr]
;        out     [UDAT]          ; Transmit data
        jmp     [lr]
```