EOR #$FF

6502 Ponderables and Befuddlements

David Youd

EOR #$FF: 6502 Ponderables and Befuddlements

EOR #$FF: 6502 Ponderables and Befuddlements

David Youd

Lulu Press, Inc.

2023

First Printing: 2023
ebook edition, version 1.0.1

Lulu Press, Inc.
700 Park Offices Drive, Suite 250
Research Triangle, NC 27709, US

draw your circle big

   draw your circle wide

      draw it so that everyone can find a place inside

open up your heart

   that's where you begin

      to draw love's circle wide enough to circle others in

"Draw Your Circle (Song)" from *Circle Sam and Other Tales* (2017)

by Jack Pearson (1953–2017), wandering minstrel, friend

Preface

As I write, Apple is selling an M2 Ultra processor that has 134 billion transistors. In contrast, the humble 6502, released in 1975, has a mere 3,510. To the programmer, the 6502 offers bottom-up simplicity, without limiting top-down expressivity. And therein lies the appeal of 8-bit retro-computing: the hardware, firmware, and software can be completely understood and reshaped by motivated hobbyists -- a microcosm for creativity.

In 2014, pseudonymous author xorpd published a paperback entitled *xchg rax,rax* (ISBN-13 978-1502958082) and also made it freely available online. It is a collection of sixty-four short x86 assembly language excerpts, one per page (numbering 0x00 to 0x3f), and presented without explanation. Its assemblage of code fragments is a masterwork in minimalism. Without guideposts, the assembly-knowledgeable reader is left to ponder the significance of each increasingly-befuddling snippet. Even the title "*xchg rax,rax*" invites consideration (it's a well-known "synthetic instruction" for NOP).

I have attempted in this book to show the same love for the 6502 chip. I've included sixty-four examples -- some original and some drawn from blogs and forums.

If you, dear reader, have not yet written any 6502 machine language code yourself, I encourage you to do so before exploring much further. There are excellent contemporary 6502 machine language learning resources, including the *Easy 6502* interactive tutorial (https://skilldrick.github.io/easy6502/). Or if you have a favorite 6502-based 8-bit machine, search archive.org for one of the many 1980s machine language books tailored to your chosen system. After some study, if you can recognize that this book's title "*EOR #$FF*" is a replacement for a common instruction that is missing in 6502 machine language, then you're probably ready for these code-reading challenges.

For those experienced with assembly language: welcome, grab a pencil and settle in. Since the book's ponderables are presented in assembly, some consistent assembler syntax had to be selected. I chose dasm (https://dasm-assembler.github.io/), a cross-platform assembler that's been under development since 1987. If you're going to execute any of these code examples, use your favorite assembler and hardware/emulator. If you choose to build using dasm, you must start your file with "PROCESSOR 6502", and pick a starting "ORG $xxxx" address that's appropriate for your environment. If you see a label containing "ZP", that belongs to page zero, the first 256 bytes of memory.

I hope that amongst these pages you'll rediscover some forgotten paths and pick up a few new tricks along the way. While I admire xorpd's spartan x86 content, this 6502 book features a back section with commentary on each of its 8-bit offerings. Resist looking at these prematurely; for recreational thinking, while not simple, is itself a simple pleasure, and in pleasure we often discover a sense of self.

$00

Category: 6502

```
EOR #VAL_1
BEQ PT1
```

$01

Category: 6502

```
         LDA ADDR16
         BNE PT1
         DEC ADDR16+1
PT1:     DEC ADDR16
```

$02

Category: 6502

```
LDA ADDR_1   |  LDA ADDR_1
CMP ADDR_2   |  CMP ADDR_2
BCC PT1      |  BMI PT1
BEQ PT1      |  BEQ PT1
```

$03

Category: Apple II

```
         LDY #$C0
PT1:     LDA #$0C
         JSR PT2
         LDA $C030
         DEY
         BNE PT1
         RTS

. . .

PT2:     SEC
PT3:     PHA
PT4:     SBC #$01
         BNE PT4
         PLA
         SBC #$01
         BNE PT3
         RTS
```

$04

Category: 6502

```
EOR #$20
```

$05

Category: Atari 2600

```
         SEI
         CLD
         LDX #$FF
         TXS
         INX
         TXA
PT1:     STA $00,X
         INX
         BNE PT1
```

$06

Category: 6502

```
EOR #$FF
SEC
ADC #$00
```

$07

Category: 6502

```
PT1:      CMP #$01
          ADC #$00
          JMP PT1
```

$08

Category: 6502

```
ROL
EOR #$01
ROR
```

$09

Category: 6502

```
          BIT ADDR
          . . .
ADDR:     RTS
```

$0A

Category: NES

```
PLA
STA $2007
PLA
STA $2007
PLA
STA $2007
. . .
PLA
STA $2007
```

$0B

Category: 6502

```
ASL
ADC #$80
ROL
ASL
ADC #$80
ROL
```

$0C

Category: 6502

```
         LDA ADDR
         INC ADDR
         CMP ADDR
         BEQ PT1
         .BYTE $02
PT1:     . . .
```

$0D

Category: 6502

```
CMP #$80
ROR
```

$0E

Category: 6502

```
PHA
LSR
PLA
ROR
```

$0F

Category: 6502

```
NOP
;
PHA
;
PLA
;
JMP (ADDR)
;
RTS
;
BRK
```

$10

Category: 6502

```
         LDA ADDR
         PHP
         LSR
         PLP
         BCC PT1
         ORA #$80
PT1:     STA ADDR
```

$11

Category: 6502

```
PTA:     STA ADDR
         LDX #$07
PT1:     LSR ADDR
         ROL
         DEX
         BPL PT1
         RTS

PTB:     PHA
         AND #$0F
         TAX
         PLA
         LSR
         LSR
         LSR
         LSR
         TAY
         LDA ADDR_1,X
         ORA ADDR_2,Y
         RTS
ADDR_1:
   .BYTE $00,$80,$40,$C0,$20,$A0,$60,$E0
   .BYTE $10,$90,$50,$D0,$30,$B0,$70,$F0
ADDR_2:
   .BYTE $00,$08,$04,$0C,$02,$0A,$06,$0E
   .BYTE $01,$09,$05,$0D,$03,$0B,$07,$0F
```

$12

Category: Atari 8-bit / Commodore 64

```
; $0278: ATARI 8-BIT
; $DC01: COMMODORE 64
IO_ADDR = $DC01

         LDA IO_ADDR
         LDY #0
         LDX #0
         LSR
         BCS PT1
         DEY
PT1:     LSR
         BCS PT2
         INY
PT2:     LSR
         BCS PT3
         DEX
PT3:     LSR
         BCS PT4
         INX
PT4:     RTS
```

$13

Category: 6502

```
ADDR_1:
         .BYTE $00,$01,$02,$03,$04,$05,$06,$07
         .BYTE $08,$09,$0A,$0B,$0C,$0D,$0E,$0F
         .BYTE $10,$11,$12,$13,$14,$15,$16,$17
                                 . . .
         .BYTE $F8,$F9,$FA,$FB,$FC,$FD,$FE,$FF

; LDY ADDR_1,X / LDX ADDR_1,Y
; AND ADDR_1,X / AND ADDR_1,Y
; ORA ADDR_1,X / ORA ADDR_1,Y
; EOR ADDR_1,X / EOR ADDR_1,Y
; ADC ADDR_1,X / ADC ADDR_1,Y
; SBC ADDR_1,X / SBC ADDR_1,Y
; CMP ADDR_1,X / CMP ADDR_1,Y
```

$14

Category: 6502

```
ASL ADDR_1
ROL ADDR_1+1
ASL ADDR_1
ROL ADDR_1+1
ASL ADDR_1
ROL ADDR_1+1
LDA ADDR_1
STA ADDR16_2
LDA ADDR_1+1
STA ADDR16_2+1
ASL ADDR_1
ROL ADDR_1+1
ASL ADDR_1
ROL ADDR_1+1
CLC
LDA ADDR_1
ADC ADDR16_2
STA ADDR_1
LDA ADDR_1+1
ADC ADDR16_2+1
STA ADDR_1+1
```

$15

Category: 6502

```
         PHA
         TXA
         TSX
         PHA
         LDA $102,X
         AND #$10
         BEQ PT1
         . . .
         JMP PT2
PT1:     . . .
PT2:     PLA
         TAX
         PLA
         RTI
```

$16

Category: 6502

```
        LDA ADDR_1
        AND ADDR_2
        STA ADDR_3
        LDA ADDR_1
        ORA ADDR_2
        SEC
        SBC ADDR_3
        RTS
ADDR_1: .BYTE  $87
ADDR_2: .BYTE  $1A
ADDR_3: .BYTE  $CF
```

$17

Category: 6502

```
LDA ADDR+1,X
PHA
LDA ADDR,X
PHA
PHP
RTI
```

$18

Category: Assembler

```
IF (PT1 & $FF00) != (PT2 & $FF00)
   ECHO "ERROR: CROSSING GUARD"
ENDIF
```

$19

Category: 6502

```
         JSR PT1
         TSX
         LDA $100,X
PT1:     RTS
```

$1A

Category: 6502

```
         JSR PT1
PT1:     PLA
         TAY
         PLA
```

$1B

Category: Various

```
         JSR PT1
         DC "SQUEAMISH OSSIFRAGE",0
         RTS

         . . .

; $FFE3: ACORN ELECTRON / BBC MODEL B
; $FDED: APPLE II
; $F6A4: ATARI 8-BIT
; $FFD2: COMMODORE 8-BIT (PET, VIC-20, C64, PLUS/4,
                          C16, C116, C128, AND C65)
; $CC12: ORIC-1 (V1.0)

ROM_CALL = $FFD2

PT1:     PLA
         STA ZP_ADDR16
         PLA
         STA ZP_ADDR16+1
PT2:     LDY #$01
         LDA (ZP_ADDR16),Y
         INC ZP_ADDR16
         BNE PT3
         INC ZP_ADDR16+1
PT3:     ORA #$00
         BEQ PT4
         JSR ROM_CALL
         JMP PT2
PT4:     LDA ZP_ADDR16+1
         PHA
         LDA ZP_ADDR16
         PHA
         RTS
```

$1C

Category: 6502

```
         LDA PT2,X
         STA PT1+1
PT1:     BNE PT1
PT2:     .BYTE VAL_1, VAL_2, VAL_3 ; ...
```

$1D

Category: 6502

```
PT1:      LDA ADDR_1
          CMP ADDR_2
          BEQ PT3
          BCS PT2
          LDX ADDR_2
          STX ADDR_1
          STA ADDR_2
          TXA
          SEC
PT2:      SBC ADDR_2
          STA ADDR_1
          BCS PT1
PT3:      RTS
```

$1E

Category: 6502

```
SED
CMP #$0A
ADC #$30
CLD
```

$1F

Category: Oric-1

```
PT1:     LDA $C000
         ROR
PT2:     LDA #'/   ; DASM FOR '/'
         BCC PT3
         EOR #%01110011
PT3:     JSR $CC12
         INC PT1+1
         BNE PT1
         INC PT1+2
         BNE PT1
         LDA #%01110011
         EOR PT2+1
         STA PT2+1
         LDA #$C0
         STA PT1+2
         BMI PT1
```

$20

Category: 6502

```
; $00 < VAL_1 < VAL_2 < $FF

PTA:     CLC
         ADC #$FF - VAL_2
         ADC #VAL_2 - VAL_1 + 1
         RTS

PTB:     SEC
         SBC #VAL_1
         SBC #VAL_2 - VAL_1 + 1
         RTS
```

$21

Category: 6502

```
         SEC
         LDA ADDR32_1+1
         SBC ADDR32_2+1
         BVC PT1
         EOR #$80
PT1:     BMI PT3
         BVC PT2
         EOR #$80
PT2:     BNE PT4
         LDA ADDR32_1
         SBC ADDR32_2
         BCC PT3
```

$22

Category: 6502

```
LDA ADDR_1
EOR ADDR_2
STA ADDR_1
EOR ADDR_2
STA ADDR_2
EOR ADDR_1
STA ADDR_1
```

$23

Category: 6502

```
         STA ZP_ADDR
         LDA #$80
PT1:     BIT ZP_ADDR
         BNE PT2
         LSR
         BCC PT1
PT2:     RTS
```

$24

Category: Various

```
        STX ZP_ADDR16
        STY ZP_ADDR16+1
        LDY #0
PT1:    LDA (ZP_ADDR16),Y
        BEQ PT6
        CMP #$7B
        BCS PT5
        CMP #$41
        BCC PT5
        CMP #$4E
        BCC PT2
        CMP #$5B
        BCC PT3
        CMP #$61
        BCC PT5
        CMP #$6E
        BCS PT3
PT2:    ADC #$0D
        BNE PT4
PT3:    SEC
        SBC #$0D
PT4:    STA (ZP_ADDR16),Y
PT5:    INY
        BNE PT1
        INC ZP_ADDR16+1
        JMP PT1
PT6:    RTS
```

$25

Category: 6502

TAX
;
TAY
;
TSX
;
TXA
;
~~TXS~~
;
TYA

$26

Category: Atari 8-bit / Commodore 8-bit

```
PT1:  CMP #$20         |  PT1:  CMP #$20
      BCC PT3          |        BCC PT5
      CMP #$60         |        CMP #$40
      BCC PT2          |        BCC PT6
      CMP #$80         |        CMP #$60
      BCC PT4          |        BCC PT4
      CMP #$A0         |        CMP #$80
      BCC PT3          |        BCC PT2
      CMP #$E0         |        CMP #$A0
      BCS PT4          |        BCC PT4
PT2:  SBC #$1F         |        CMP #$C0
      RTS              |        BCC PT3
PT3:  ORA #%01000000   |        CMP #$FF
PT4:  RTS              |        BCC PT5
                       |        LDA #$5E
                       |        RTS
                       |  PT2:  AND #%11011111
                       |        RTS
                       |  PT3:  EOR #%11000000
                       |        RTS
                       |  PT4:  EOR #%01000000
                       |        RTS
                       |  PT5:  EOR #%10000000
                       |  PT6:  RTS
```

$27

Category: 6502

```
LDA ADDR_1
EOR ADDR_2
AND ADDR_3
BEQ PT1
```

$28

Category: 6502

```
PTA:      EOR ADDR
          AND #%00111100
          EOR ADDR
          STA ADDR
          RTS

PTB:      EOR ADDR
          AND #%11000011
          EOR ADDR
          RTS
```

$29

Category: 6502

```
         LDX #0
         LDA ADDR_1
         BEQ PT2
PT1:     INX
         DEC ADDR_1
         AND ADDR_1
         STA ADDR_1
         BNE PT1
PT2:     RTS
```

$2A

Category: 6502

```
ORG $0900

LDY #$0E       ; 0900 A0 0E
LDA $0901,Y    ; 0902 B9 01 09
EOR $0900,Y    ; 0905 59 00 09
STA $0901,Y    ; 0908 99 01 09
INY            ; 090B C8
CPY #$B7       ; 090C C0 B7
BNE $0932      ; 090E D0 22
.BYTE $D2,$C7 ; 0910 D2 C7
CLC            ; 0912 18
LSR $A9,X      ; 0913 56 A9
. . .
```

$2B

Category: Various

```
        LDA $FFFA
        CMP #$18
        BNE PT2
PT1:    INC $02c8
        JMP PT1
PT2:    CMP #$43
        BNE PT4
PT3:    INC $D020
        JMP PT3
PT4:    CMP #$05
        BEQ PT3
        CMP #$A4
        BNE PT6
PT5:    INC $FF19
        JMP PT5
PT6:    CMP #$A9
        BNE PT8
        LDA $900F
        AND #$F8
        STA PT8
PT7:    INX
        TXA
        AND #$07
        ORA PT8
        STA $900F
        JMP PT7
PT8:    RTS
```

$2C

Category: 6502

```
PTA:     STA ADDR_1
         LSR ADDR_1
         EOR ADDR_1
         RTS

PTB:     STA ADDR_1
PT1:     LDY ADDR_1
         BEQ PT2
         LSR ADDR_1
         EOR ADDR_1
         JMP PT1
PT2:     RTS
```

$2D

Category: 6502

```
PT1:     LDX #$44
         .BYTE $2C
PT2:     LDX #$57
         .BYTE $2C
PT3:     LDX #$59
```

$2E

Category: Apple II

```
PT1:
  .BYTE $15,$93,$17,$A3,$35,$CE,$6C,$10,$B9,$66,$19,$D0
  .BYTE $8B,$49,$0C,$D2,$9B,$67,$36,$08,$DC,$B3,$8C,$68
  .BYTE $45,$25,$06,$E9,$CD,$B3,$9B,$84,$6E,$5A,$46,$34
  .BYTE $23,$12,$03,$F4,$E7,$DA,$CE,$C2,$B7,$AD,$A3,$9A
  .BYTE $91,$89,$81,$7A,$73,$6D,$67,$61,$5C,$56,$52,$4D
  .BYTE $49,$45,$41,$3D,$3A,$36,$33,$30,$2E,$2B,$29,$26
  .BYTE $24,$22,$20,$1F,$1D,$1B,$1A,$18,$17,$16,$14,$13
  .BYTE $12,$11,$10,$0F,$0E,$0E,$0D,$0C,$0B,$0B,$0A,$0A

PT2:
  .BYTE $09,$08,$08,$07,$07,$06,$06,$06,$05,$05,$05,$04
  .BYTE $04,$04,$04,$03,$03,$03,$03,$03,$02,$02,$02,$02
  .BYTE $02,$02,$02,$01,$01,$01,$01,$01,$01,$01,$01,$01
  .BYTE $01,$01,$01,$00,$00,$00,$00,$00,$00,$00,$00,$00
  .BYTE $00,$00,$00,$00,$00,$00,$00,$00,$00,$00,$00,$00
  .BYTE $00,$00,$00,$00,$00,$00,$00,$00,$00,$00,$00,$00
  .BYTE $00,$00,$00,$00,$00,$00,$00,$00,$00,$00,$00,$00
  .BYTE $00,$00,$00,$00,$00,$00,$00,$00,$00,$00,$00,$00
```

$2F

Category: 6502

```
PT1:      LDX #$00
          TXS
          LDA #>PT1
          PHA
          LDA #<PT1
          PHA
          JMP ($01FF)
```

$30

Category: 6502

```
         LDY #16
         JSR PT1
         RTS

PT1:     LDA ADDR16
PT2:     ASL
         ROL ADDR16+1
         BCC PT3
         EOR #%00111001
PT3:     DEY
         BNE PT2
         STA ADDR16
         RTS

ADDR16:  .BYTE $AA,$AA
```

$31

Category: 6502

```
         STA ADDR
         SED
         LDX #8
PT1:     ASL ADDR
         LDA ADDR16
         ADC ADDR16
         STA ADDR16
         LDA ADDR16+1
         ADC ADDR16+1
         STA ADDR16+1
         DEX
         BNE PT1
         CLD
         RTS
ADDR16:  .BYTE $00,$00
```

$32

Category: Atari 2600

```
         . . .

         BRK
         .BYTE VAL1

         . . .

PT1:     PLP
         TSX
         INX
         DEC $00,X
         LDA ($00,X)
         . . .
         RTS

         . . .

         ORG $FFFE
         .BYTE <PT1, >PT1
```

$33

Category: 6502

```
CMP #$C9
CMP #$C9
CMP #$C9
CMP #$C9
CMP #$C9
CMP #$C9
CMP $EA
```

$34

Category: 6502

```
STA ADDR
LSR
ADC #$15
LSR
ADC ADDR
ROR
LSR
ADC ADDR
ROR
LSR
ADC ADDR
ROR
LSR
```

$35

Category: 6502

```
         LDA #$40
PT1:     .BYTE $CF, <ADDR_1, >ADDR_1
         BNE PT1
         RTS
ADDR_1:  .BYTE $5A
```

$36

Category: 6502

```
          LDA VAL    |
          BPL PT1    |  PT1:     CMP #$41
          AND #$7F   |           BCC PT2
          JSR PT1    |           EOR #$7F
          EOR #$FF   |           ADC #$00
          CLC        |  PT2:     ASL
          ADC #$01   |           TAX
          PHA        |           LDA PT3,X
          TXA        |           PHA
          EOR #$FF   |           LDA PT3+1,X
          ADC #$00   |           TAX
          TAX        |           PLA
          PLA        |           RTS
          RTS        |

PT3:
  .WORD $0000,$0324,$0647,$096A,$0C8B,$0FAB,$12C8,$15E2
  .WORD $18F8,$1C0B,$1F19,$2223,$2528,$2826,$2B1F,$2E11
  .WORD $30FB,$33DE,$36BE,$398C,$3C56,$3F17,$41CE,$447A
  .WORD $471C,$49B4,$4C3F,$4EBF,$5133,$539B,$55F5,$5842
  .WORD $5A82,$5CB4,$5ED7,$60EC,$62F2,$64EB,$66CF,$68A6
  .WORD $6A6D,$6C24,$6DC4,$6F5F,$70E2,$7255,$73B5,$7504
  .WORD $7641,$776C,$7884,$798A,$7A7D,$7B5D,$7C2A,$7CE3
  .WORD $7D8A,$7E1D,$7E9D,$7F09,$7F62,$7FA7,$7FD8,$7FF6
  .WORD $7FFF
```

$37

Category: 6502

```
PT1:     STX ADDR            |           STA ADDR32+1
         STA ZP_ADDR16+1     |           LDA #$20
         LDY #0              |           EOR ADDR32+0
         STY ZP_ADDR16       |           STA ADDR32+0
         CLC                 |  PT4:     DEX
         JSR PT5             |           BNE PT3
PT2:     LDA (ZP_ADDR16),Y   |           INY
         LDX #8              |           BNE PT2
         EOR ADDR32+0        |           INC ZP_ADDR16+1
         STA ADDR32+0        |           DEC ADDR
PT3:     LSR ADDR32+3        |           BNE PT2
         ROR ADDR32+2        |           SEC
         ROR ADDR32+1        |           JSR PT5
         ROR ADDR32+0        |           RTS
         BCC PT4             |  PT5:     LDX #3
         LDA #$ED            |  PT6:     LDA #$FF
         EOR ADDR32+3        |           BCC PT7
         STA ADDR32+3        |           EOR ADDR32+0,X
         LDA #$B8            |  PT7:     STA ADDR32+0,X
         EOR ADDR32+2        |           DEX
         STA ADDR32+2        |           BPL PT6
         LDA #$83            |           RTS
         EOR ADDR32+1        |
```

$38

Category: 6502/65C02

```
SED
CLC
LDA #$99
ADC #$01
CLD
```

$39

Category: 6502

```
         CPX #$03
         BCS PT2
         CPY #$00
         BNE PT1
         LDY #$06
PT1:     DEY
PT2:     EOR #%01111111
         CPY #$C8
         ADC PT4,X
         STA PT4
         TYA
         JSR PT3
         SBC PT4
         STA PT4
         TYA
         LSR
         LSR
         CLC
         ADC PT4
PT3:     ADC #7
         BCC PT3
         ADC #$00
         RTS
PT4:     .BYTE 0,1,5,6,3,1,5,3,0,4,2,6,4
```

$3A

Category: 6502

```
ZP_ADDR_1     .BYTE $07,$07,$07,$07,$07,$07,$07,$07
ZP_ADDR_2     .BYTE $FF,$FF,$FF,$FF,$FF,$FF,$FF,$07

. . .

PT1:     LDX #7              |          BEQ PT7
PT2:     LDA ZP_ADDR_1,X     |          LDA #$00
         STX ADDR_1          |          STA ADDR_2
         TAX                 |          TXA
         DEC ZP_ADDR_2,X     |          TAY
         LDX ADDR_1          | PT6:     DEY
         DEC ZP_ADDR_1,X     |          BMI PT5
         BPL PT3             |          LDA ZP_ADDR_1,Y
         LDY #7              |          SEC
         STY ZP_ADDR_1,X     |          SBC ZP_ADDR_1,X
         INC ZP_ADDR_2+7     |          INC ADDR_2
         DEX                 |          CMP ADDR_2
         BPL PT2             |          BEQ PT1
         BMI PT8             |          EOR #$FF
PT3:     TAX                 |          SEC
         INC ZP_ADDR_2-1,X   |          ADC #$00
         LDX #7              |          CMP ADDR_2
PT4:     LDY ZP_ADDR_2,X     |          BNE PT6
         BNE PT1             |          BEQ PT1
         DEX                 | PT7:     CLC
         BPL PT4             |          RTS
         LDX #8              | PT8:     SEC
PT5:     DEX                 |          RTS
```

$3B

Category: 6502

```
PTA:     LDX #$FF    |            BMI PT6
         LDA ADDR_2  |            BEQ PT6
         SEC         |            LDA ADDR_9
         SBC ADDR_4  |            SEC
         BPL PT1     |            SBC ADDR_6
         LDX #$01    |            STA ADDR_9
         JSR PT5     |            LDA ADDR_1
PT1:     STA ADDR_6  |            CLC
         STX ADDR_8  |            ADC ADDR_7
         LDX #$FF    |            STA ADDR_1
         LDA ADDR_1  | PT6:       PLA
         SEC         |            CMP ADDR_6
         SBC ADDR_3  |            BPL PT7
         BPL PT2     |            LDA ADDR_9
         LDX #$01    |            CLC
         JSR PT5     |            ADC ADDR_5
PT2:     STA ADDR_5  |            STA ADDR_9
         STX ADDR_7  |            LDA ADDR_2
         CMP ADDR_6  |            CLC
         BEQ PT3     |            ADC ADDR_8
         BPL PT4     |            STA ADDR_2
PT3:     LDA ADDR_6  | PT7:       LDA ADDR_1
         JSR PT5     |            CMP ADDR_3
PT4:     STA ADDR_9  |            BNE PT8
         ASL ADDR_5  |            LDA ADDR_2
         ASL ADDR_6  |            CMP ADDR_4
         RTS         |            BEQ PT9
PT5:     EOR #$FF    | PT8:       CLC
         CLC         | PT9:       RTS
         ADC #$01    | ADDR_5:    BRK
         RTS         | ADDR_6:    BRK
PTB:     LDA ADDR_9  | ADDR_7:    BRK
         PHA         | ADDR_8:    BRK
         CLC         | ADDR_9:    BRK
         ADC ADDR_5  |
```

$3C

Category: 6502

```
PT0:
   .BYTE $97,$49,$20,$44,$4F,$4E,$27,$54 ; —I DON'T
   .BYTE $20,$4B,$4E,$4F,$57,$20,$48,$41 ;  KNOW HA
   .BYTE $4C,$46,$20,$4F,$46,$20,$59,$4F ; LF OF YO
   .BYTE $55,$27,$F4,$86,$41,$53,$20,$57 ; U'ô†AS W
   .BYTE $45,$4C,$4C,$17,$F8,$D6,$8E,$53 ; ELL.øÖŽS
   .BYTE $48,$4F,$55,$4C,$44,$20,$4C,$49 ; HOULD LI
   .BYTE $4B,$45,$20,$41,$4E,$44,$ED,$27 ; KE ANDí'
   .BYTE $F5,$86,$4C,$45,$53,$53,$20,$54 ; õ†LESS T
   .BYTE $48,$ED,$7F,$C1,$57,$C1,$17,$EC ; Hí.ÁWÁ.ì
   .BYTE $86,$44,$45,$53,$45,$52,$56,$45 ; †DESERVE
   .BYTE $00
PT1:  LDA #<PT8            |        BEQ PT8
      STA ZP_ADDR16_1      |        ASL
      LDA #>PT8            |        BCC PT6
      STA ZP_ADDR16_1+1    |        BPL PT2
      LDA #<PT0            |        LDX #$07
      STA ZP_ADDR16_2      |        .BYTE $F0 ; BEQ NOP
      LDA #>PT0            |  PT6:  INY
      STA ZP_ADDR16_2+1    |        LDA (ZP_ADDR16_2),Y
      JMP PT5              |        ADC ZP_ADDR16_1
PT2:  INY                  |        STA ZP_ADDR16_3
      LDA (ZP_ADDR16_2),Y  |        TXA
      STA (ZP_ADDR16_1),Y  |        ORA #$F8
      DEX                  |        ADC ZP_ADDR16_1+1
      BMI PT2              |        STA ZP_ADDR16_3+1
      TYA                  |        TYA
      PHA                  |        PHA
      CLC                  |        LDY #1
PT3:  ADC ZP_ADDR16_1      |        LDA (ZP_ADDR16_3),Y
      STA ZP_ADDR16_1      |        STA (ZP_ADDR16_1),Y
      BCC PT4              |  PT7:  INY
      INC ZP_ADDR16_1+1    |        LDA (ZP_ADDR16_3),Y
PT4:  PLA                  |        STA (ZP_ADDR16_1),Y
      SEC                  |        TXA
      ADC ZP_ADDR16_2      |        SBC #$08
      STA ZP_ADDR16_2      |        TAX
      BCC PT5              |        BPL PT7
      INC ZP_ADDR16_2+1    |        TYA
PT5:  LDY #0               |        BCC PT3 ; ALWAYS
      LDA (ZP_ADDR16_2),Y  |  PT8:  RTS
      TAX                  |
```

$3D

Category: 6502

```
; AT $8000, BUT A BIT OFF...
8014   8E 32 A0   STX $A032
8017   8D 33 A0   STA $A033
801A   7D CA 5F   ADC $5FCA,X
801D   BD DE 5F   LDA $5FDE,X
8020   EA         NOP
8021   C0 35      CPY #$35
8023   F0 D2      BEQ $7FF7
8025   40         RTI
8026   09 00      ORA #$00
```

$3E

Category: 6502

```
        ORG $1000

        SEI       ; 2
        SEC       ; 2
        LDY #253  ; 2
        LDX #255  ; 2
PT1:    LDA #44   ; 2*255*253
PT2:    SBC #1    ; ETC.
        BNE PT2   ; CYCLES TO DECODE...
        DEX
        BNE PT1
        LDX #255
        LDA #12
PT3:    SBC #1
        BNE PT3
        BIT PT3
        DEY
        BNE PT1
        RTS
```

$3F

Category: 6502

```
         ORG $4040

         DC ")@)?8iAH8izipi!Hiziai!Hi20$hHizi]i!LG@"

; AS BYTES:
; $29, $40, $29, $3F, $38, $69, $41, $48
; $38, $69, $7A, $69, $70, $69, $21, $48
; $69, $7A, $69, $61, $69, $21, $48, $69
; $32, $30, $24, $68, $48, $69, $7A, $69
; $5D, $69, $21, $4C, $47, $40
```

Commentary

$00 Category: 6502

Perform an equality check without using CMP; unlike using CMP, this approach leaves the carry flag alone. It modifies the accumulator value (which could be restored with a second EOR #VAL_1).

$01 Category: 6502

A 16-bit decrement.

$02 Category: 6502

Branch if ADDR_1 < = ADDR_2. The example on the right is the same, but for signed values.

Note: An instruction can be removed if one flips the order of the comparators; i.e., the left example could be rewritten as:

```
LDA ADDR_2
CMP ADDR_1
BCS PT1
```

$03 Category: Apple II

Produce a tone on an Apple II. Reading from or writing to (strobing) $C030 toggles the flip-flop to the speaker, reversing the flow of current in the coil, moving the speaker cone either in or out. Two such toggles complete a square-wave period, and the toggle rate determines pitch (audible if from 20 to 20,000Hz).

These code examples come directly from the Apple II ROM that produces the system beep via a bell routine ($FBD9) and a wait routine ($FCA8). The inner-accumulator loop sets the pitch, while the outer-Y-register loop sets the duration. On an NTSC Apple II, this produces a ~1KHz tone (a B5) that lasts about a 10th of a second.

$04 Category: 6502

Toggles the upper/lower case of a letter. Commodore 8-bit, Atari 8-bit, Acorn Electron, BBC Micro, ORIC-1/Atmos, and other 6502-based machines use ASCII-like character encodings. These machines all feature alphabet ranges offset by 32 from another opposite-case alphabet range.

Note: The book *xchg rax,rax* uses this example (in x86 assembly) for problem 0x04.

$05 Category: Atari 2600

Standard Atari 2600 game cartridge boilerplate for initializing its hardware that starts in an undefined state. First, interrupts are turned off (a common practice in 2600 game carts, even though the 6507 processor has no IRQ or NMI pins). Decimal mode is cleared and the top of the stack is set (to empty). Page zero is cleared, specifically, the TIA (Television Interface Adaptor) registers from $00 to $2C and the 128 bytes of RAM from $80 to $FF.

$06 Category: 6502

Multiply accumulator by -1, aka, perform a two's complement. Performed by inverting and adding 1. Example, 2 becomes -2:

```
        %00000010 -> %11111101 -> %11111110
```

$07 Category: 6502

This infinite loop will count up from a given value, wrap, and cease incrementing once reaching zero. The compare with 1 will set the carry only if the accumulator > 0, and only if the carry is set will the ADC #$00 increment.

$08 Category: 6502

Toggles the carry. Is nondestructive to the accumulator value, but affects the N and Z flags. This example is from Lee Davidson's collection of very short code bits (Davidson's website content recovered / reconstructed by Hans Otten).

$09 Category: 6502

The 6502 has a CLV instruction to clear the oVerflow flag, but there is no corresponding instruction to set the flag. Performing BIT on an RTS is a synthetic instruction that sets the V flag. (A synthetic instruction is either an assembler alias for one or more instructions, or as in this case, is one or more instructions that represent an unimplemented instruction).

The BIT instruction takes an 8-bit (ZP) or 16-bit memory location, and transfers bit 7 (the left-most bit) and bit 6 of the value at that address into the N and V flags respectively (bits 7 and 6 of the status register). Since the RTS opcode is %01100000, this code example sets the V flag. RTS instructions are common, so there is usually one nearby to press into service.

Another way to set the oVerflow flag is via pin 38 (SO) on the 6502 itself. Most Commodore-designed floppy drives set the V flag in this way (via pin 38) when a byte of data has finished being read (flux to GCR) or written (GCR to flux). This is why the Commodore 1541 floppy disk drive ROM contains a few dozen examples of this code pattern:

```
PT1:     BVC PT1
         CLV
```

Nate Lawson explains: *"This pin was also labeled CPS ('Chuck Peddle Special') in the earliest versions of the 6502 since the designer was an advocate for this behavior. Chuck imagined the 6502 as a cheap microcontroller for embedded and industrial devices, and this was a convenient way to interface with an external peripheral. I haven't seen any other devices use this pin other than Commodore disk drives..."*

Notes:

– The BIT instruction also ANDs together the accumulator and memory, setting the Z flag on the result (then tossing that result).

– Until the advent of the 65C02, the BIT instruction didn't support immediate addressing, but you can fake it by pointing the operand address to a known byte value; i.e., for a Z-flag test of bit 6, choose an address holding an RTI (RTI is %01000000), for bit 5 a JSR, bit 4 a BPL, and for bit 3 a PHP.

$0A Category: NES

A technique some Nintendo games use to quickly move data to the PPU (Picture Processing Unit). The PPU has exclusive access to video memory while it sends a picture to the screen. The CPU may access video memory during the relatively-short vblank vertical blanking interval (or by forcing blanking). Games can be designed to buffer their video updates before rapidly sending the collected data to the PPU (i.e., the $2007 VRAM read/write register) during a vblank. A common place to buffer is in the stack. An unrolled STA loop of PLA / STA $2007 pairs is more space efficient than LDA ADDR+offset / STA $2007 pairs, and consumes the same number of cycles.

$0B Category: 6502

Swaps nybbles in the accumulator value, using only the status register. Referenced by David Galloway on the Facebook 6502 Programming group.

$0C Category: 6502

Executing $02 will freeze the processor. Copy protection programs sometimes snuck such a "JAM" instruction ($02) into the execution path if tampering (or a copy) was suspected. This JAM was sometimes executed if a checksum over the code bytes didn't give the expected result.

In this example, the code assumes it's running in ROM (i.e., a game cartridge), and will crash if it detects that it can change what should be read-only memory. Note: this approach will crash an NMOS

6502 but not a 65C02 system (e.g., Apple IIe enhanced or GameKing I/II) or 65SC02 system (e.g., Watara Supervision or Atari Lynx).

$0D Category: 6502

Perform integer division by 2 for a signed number. Here's a visual reminder of the number line for 8-bit signed values:

```
... -3 -2 -1 +0 +1 +2 +3 ...
    FD FE FF 00 01 02 03
```

While an LSR-based divide-by-2 rounds down towards zero (e.g., 5 becomes 2), for negative numbers, this approach also rounds down, but away from zero (e.g., -5 becomes -3).

$0E Category: 6502

Perform an 8-bit rotate right (in contrast to the ROR, a 9-bit rotate right through the carry).

Note: Copy protection on the Apple II version of *The Goonies* (Datasoft, 1985) used this sequence for decryption.

$0F Category: 6502

Lateral thinking required (not reading of control flow). The instructions are in increasing cycle-count order, from 2 cycles to 7.

$10 Category: 6502

Early 6502 chips had a broken ROR instruction. This code example is a proposed ROR workaround described in the November 1975 issue of *BYTE Magazine* (page 58, table II).

$11 Category: 6502

Two examples that reverse the order of bits. The first requires fewer bytes and 109 cycles; the second is larger, but only consumes 29 cycles. Of course, the fastest approach would be a single lookup instruction into a table of 256 precomputed values. There's plenty of opportunity for readers to find smaller, faster, worst-case faster, and/or constant-time versions of all the examples in this book.

$12 Category: Atari 8-bit / Commodore 64

Read the joystick up/down into Y (-1, 0, 1) and left/right into X (-1, 0, 1). Reading the C64 joystick 1 or the Atari 8-bit stick 0 will produce the same lower nybble values for the same joystick directions:

	C64	Atari 8-bit
	$DC01	$0278
Up:	%11111110	%00001110
Down:	%11111101	%00001101
Left:	%11111011	%00001011
Right:	%11110111	%00000111

The Atari 400/800 joystick I/O is handled by its MOS MCS6520 Peripheral Interface Adapter (PIA) chip. The Commodore 64 uses the MOS 6526/8520 Complex Interface Adapter (CIA) chip, a descendant of the PIA.

As with many of the examples in this book, using the computer's screen memory is a great way to test. In this case, build an infinite loop that calls the joystick read routine and does an STX $0400 / STY $0401 (C64) or STX $9C40 / STY $9C41 (Atari 8-bit). The two top-left-most characters on the display will reflect the joystick direction, showing screen code symbols for 1, 0, or -1 (255).

$13 Category: 6502

These additional "synthetic instructions" are available if you can afford a page (256 bytes) of constants. It's like having TYX, TXY, AND X, AND Y, ORA X/Y, EOR X/Y, ADC X/Y, SBC X/Y, and CMP X/Y. You can also point BIT directly to a table entry with a desired bit test pattern, as if BIT took an immediate value.

$14 Category: 6502

This takes the single byte in ADDR_1, multiplies by 40, and puts the 16-bit result in ADDR_2. Such a routine is useful in screen positioning logic on 40-column displays. val*40 was decomposed into shifts and adds:

```
   (val*8)*5 = val*8*4 + val*8 = (val<<3)<<2 + (val<<3)
```

Can you find a faster way?

In comparison, generalized multiplication routines can be much shorter. This example multiplies NUM1 * NUM2, weighs in at just 26 bytes, but requires more (and a varying number) of cycles:

```
         LDA #$80
         STA RESULT
         ASL
         DEC NUM1
PT1:     LSR NUM2
         BCC PT2
         ADC NUM1
PT2:     ROR
         ROR RESULT
         BCC PT1
         STA RESULT+1
```

Note: Toby Nelson has open-sourced a test suite that exhaustively compares over 120 6502 multiplication routines.

$15 Category: 6502

In an interrupt handler routine, determine if the source was an IRQ (hardware) or BRK (software) interrupt by reading a flags byte in the stack and testing bit 4. (It's sometimes necessary to differentiate between the two because IRQ and BRK both use the same vector at $FFFE.)

There is a subtle bug in this code: it's not safe when the stack wraps. Can you find and fix the bug? (Hint: change one instruction and add two more).

Despite what your favorite emulator or monitor (or book) may tell you, the B flag does not maintain its state in the 6502 status register. It only exists as a bit within a byte that gets pushed to the stack during an IRQ entry sequence. BRK and PHP put B = 1 on the stack, while IRQ and NMI put B = 0. A PLP or RTI instruction can restore stack-stored flags to the status register, but the B flag state has no place to go, as there's no register storage for it on the CPU. So its existence must be tested while it's still on the stack.

The belief that the break flag state exists in the status register can lead to errors. Avery Lee describes such a bug in the Atari 8-bit OS Version A (included in early NTSC Atari 400/800 systems). Its BRK handler can mistake IRQs for BRKs by pushing the status register to the stack and checking that byte in a vain attempt to get the B flag state. But the B flag doesn't have a defined register state to be pushed, and PHP always pushes bit 4 as a 1.

$16 Category: 6502

Performs exclusive-or functionality without using an EOR instruction. In pseudo code, this is done via the equation:

```
val1 XOR val2 = (val1 OR val2) - (val1 AND val2)
```

6502-based copy protection code is frequently exclusive-or obfuscated, which is why reversers often start by hunting for EOR instructions. Various methods were used to hide EOR instructions in software loaders.

$17 Category: 6502

This code will jump to the address it finds at ADDR+X (presumably a table of addresses). Had RTS been used instead of RTI (to avoid the PHP), the address would have to be one less than the intended target, since RTS adds one to the address it pulls from the stack. (A later example will use the RTS approach.)

$18 Category: Assembler

This is dasm syntax for making sure that addresses PT1 and PT2 do not fall on separate pages. This guard code makes sure that cycle-exact sections of code don't span page boundaries unexpectedly since each branch across a page boundary incurs an additional cycle.

$19 Category: 6502

Determines the page containing the executing code. After the JSR, the stack is pointing to the high byte of the address (the page) hosting the RTS. The stack index is used to load that page number into the accumulator.

Note: This technique is described in *Byte Magazine* Volume 10, issue 6 (June, 1985) in "6502 Tricks and Traps", by Joe Holt. It's also used in the Apple II boot PROM to discover which expansion slot is executing code.

$1A Category: 6502

Get the program counter. Can be used in assisting code relocation, or in preventing code relocation (from analysis, repacking, etc.).

The book *xchg rax,rax* uses a similar example (in x86 assembly) for problem 0x1a.

$1B Category: Various

Example of a JSR that "passes a string"; the string immediately follows the JSR instruction, as if it were a call parameter. The location of the start of the string is derived from the return address on the stack. RTS pulls two bytes from the stack (low first), and sets the program counter to that address + 1. On entry, the stack points to the byte before the start of the string, and on exit, the stack points to the null ($00) at the end of the string.

Even more systems to try:

– $FFD2 also works on the Commander X16 (a modern 6502-based system).
– Will work on the ORIC ATMOS (V1.1) using $F77C, but the code must be modified to put the character in the X register (not the accumulator).

$1C Category: 6502

Like a BASIC "ON X GOTO" or C switch() statement, this self-modifying code supports different branch destinations based on an X register index.

$1D Category: 6502

Implements Euclid's Greatest Common Divisor algorithm, detailed in his *Elements* (written 300 BC). When iterating, performs a swap to make sure the smaller working value is always subtracted from the larger. The SBC result is always >= 1, so the BCS that follows is an unconditional branch.

$1E Category: 6502

Converts a hex digit (0 to F) to ASCII. This four-line method was found in a decimal mode tutorial by Bruce Clark.

BCD caveat: Unlike the 65C02, the NMOS 6502 does not automatically clear decimal mode on interrupts. When an interrupt occurs, the status register (which includes the current binary or decimal mode state) is put on the stack, and then restored after the interrupt. But if an interrupt routine on an NMOS 6502 uses an ADC or SBC, it may have (unwisely) assumed that it is in binary mode (without the precaution of a CLD instruction). This happened with the Commodore 64 kernal's housekeeping interrupt handler. So if trying this example on a Commodore 64, be sure to wrap the code in a warm, safe SEI/CLI blanket.

$1F Category: Oric-1

This program creates an emergent maze-like display by generating a random-looking sequence of forward (ASCII 47) and back (ASCII 92) slash symbols. These mazes have been repopularized in the last decade by Nick Montfort's 2014 book (with the unwieldy title) *10 PRINT CHR$(205.5+RND(1)); : GOTO 10*, and the *8-bit Show and Tell* Youtube channel (where Robin Harbron reimplements the maze in various ways to demonstrate early computer languages, different 8-bit systems, and opportunities for optimization).

The code example targets an Oric-1 (v1.0), printing symbols via the routine at $CC12. It uses multiple instances of self-modifying code. The appearance of randomness comes from choosing the slash directions based upon the least significant bits of the BASIC ROM bytes from $C000 to $FFFF. The program repeatedly loops through this ROM range, and on every-other repeat, the assignment of slash directions is reversed.

Slash-maze generators that make use of system-specific memory-mapped I/O can be much smaller, as this Atari 8-bit implementation (posted by xxl in the AtariAge forums) demonstrates:

```
PT1:    ROL $D20A  ; GET RANDOM FROM AN LFSR
        LDA #$03
        ROL        ; 6='/', 7='\'
        JSR $F2B0  ; PRINT
        BCS PT1
```

Even shorter is this four-line Commodore 64 version developed by Paul Kocyla. It uses some clever stack manipulations and return-oriented programming. After launching the maze code at $2F00 with "SYS 12032", the SYS BASIC ROM routine forces address $E146 on the stack, then indirect JMPs to $2F00:

```
2F00  PLA        ; A = $E1
2F01  ROR $DC04  ; GET CARRY FROM CIA1 TIMER A LOW BYTE
2F04  ADC #$87   ; 205 ('\') OR 206 ('/')
2F06  JSR $E717  ; ENTER PRINT ROUTINE, SKIP FIRST PHA
```

The print routine finishes by returning to the part of the SYS routine that forces address $E146 on stack and then JMPs to $2F00, closing the loop.

$20 Category: 6502

Test if an unsigned byte is within the range VAL_1 to VAL_2 (inclusive). Only requires 3 instructions, but at the cost of modifying the value being tested. The 1st approach (PTA) sets the carry if the accumulator is within the range. The 2nd approach (PTB) clears the carry if the accumulator is in the range. Recall that 1) if an addition result is 0 to 255, the carry is cleared, but if greater than 255, it's set, and 2) if a subtraction result is 0 to 255, the carry is set, but if less than 0, it's cleared. Code examples found in Lee Davidson's collection of very short code bits.

$21 Category: 6502

A 16-bit signed comparison that branches to PT3 if ADDR32_1 < ADDR32_2, otherwise PT4. N, Z, and C flags work as you'd expect. Described on Bruce Clark's excellent webpage *Beyond 8-bit Unsigned Comparisons*.

$22 Category: 6502

This is an obfuscated way of swapping two values. It's based on the old (and terrible) interview question concerning how to swap two stored values without using a 3rd intermediate stored value, for which a solution is:

```
x = x xor y
y = x xor y
x = x xor y
```

$23 Category: 6502

Rounds the accumulator down to the nearest power of 2. Created by Antonio Savona and contributed here by Robin Harbron.

$24 Category: Various

Perform an in-place ROT-13 on a mixed-case, zero-terminated ASCII string.

$25 Category: 6502

Why is TXS unlike the others? It doesn't touch the flags; all the other transfer instructions set N and Z.

$26 Category: Atari 8-bit / Commodore 8-bit

The first example converts an Atari ATASCII character to its screen code. The second example does the same for Commodore PETSCII.

A screen code is the character encoding used in the RAM backing the user's text display. The characters' numeric values reflect the order of the graphic bitmap definitions found in the system's character ROM.

$27 Category: 6502

For bit positions set to 1 in ADDR_3, test that those bit positions in ADDR_1 and ADDR_2 have the same values. If so, the BEQ branch is taken.

For example, if ADDR_3 is %10101010, the code will branch if the values of bits 7, 5, 3, and 1 are the same for both ADDR_1 and ADDR_2.

$28 Category: 6502

Two routines that copy the values of specified bits. In the 1st example (PTA), the bits in the accumulator that are specified by 1s in the AND mask have their values copied into ADDR. In the 2nd example (PTB), the bits in ADDR that are specified by 0s in the AND mask have their values copied into the accumulator. While the approach generalizes to any bit selection, in these specific examples, the middle 4 bits (bits 2 through 5) are copied. Thanks to "supercat" in the AtariAge forums for this concept.

$29 Category: 6502

Count the number of bits set to 1 in the accumulator using Dr. Brian Kernighan's classic bit counting algorithm.

$2A Category: 6502

A varying-mask exclusive-or deobfuscation loop in the game loader code for the C64 version of *Robots of Dawn* (EPYX, 1984). Note that the branch that drives the loop is itself deobfuscated (repaired) before the first pass of the loop completes. In this code excerpt, the last seven bytes at $090E become $D0, $F2, $20, $E7, $FF, $A9, and $00, which is BNE $0902, JSR $FFE7, and LDA #$00.

$2B Category: Various

This code will cycle the border colors on Atari 8-bit and some Commodore 8-bit (i.e., C64, C128, VIC-20, Plus/4 (C16)) machines in platform-specific ways. It detects which system it's on by looking at the differences in the NMI vector that the 6502 expects at $FFFA. Note: changing the border color is a time-honored debugging technique akin to printing "got this far".

$2C Category: 6502

The first code snippet (PTA) converts a byte to its corresponding 8-bit reflected Gray code, and the second (PTB) reverses the process.

Gray codes are an ordering of binary numbers such that two adjacent values differ by only one bit value. There are many useful types of Gray codes, but the simplest to construct is the reflected Gray code.

$2D Category: 6502

Sometimes instruction operands are themselves valid operators; this creates opportunities for jumping into the middle of instructions. In this example, operands are also BIT instructions that function as faux NOPs to support compact branching logic. If entered at PT3, X is set to $59. If entered at PT2, X is set to $57 (LDX #$57, BIT $59A2). And if entered at P1, X becomes $44 (LDX #$44, BIT $57A2, BIT $59A2).

Notes:

– The C64 BASIC ROM made extensive use of this technique (see disassembly at $AEF9, $AEFC, $B247, $B3AD, $BAC3, and $BBC9).

– BIT is not actually a NOP, as it sets the N, V, and C flags, and if the chosen address is I/O mapped, it could create side effects.

$2E Category: Apple II

For anyone that has coded 8-bit music in assembly, low-byte and high-byte frequency tables are instantly recognizable. Mapping computer frequencies to standard pitches almost always necessitates lookup tables. Frequency is logarithmic, which is easy to see in the high-byte table values.

These particular tables target the Apple II Mockingboard sound card (Sweet Micro Systems, 1983). They were extracted by Markus Brenner from Kenneth W. Arnold's music routines in the game *Ultima III: Exodus* (Origin Systems, 1983). For each table, the columns are A, A#, B, C, C#, D, D#, E, F, F#, G, and G#, and the rows range from A0-G#1 (row 1) to A7-G#8 (row 8).

Arnold used slightly higher values (more flat) than suggested in the Mockingboard documentation, and the Mockingboard documentation also used higher values (again, more flat) than a frequency table designed to most closely match the standard 440Hz tuning (though Arnold's 440Hz A4 is, itself, set at the ideal Mockingboard value of $0091).

$2F Category: 6502

An infinite loop. The stack spans page 1, growing backwards from $01FF to $0100. LDX #$00, TXS moves the stack pointer to the topmost index ($0100), where the high byte of address PT1 is pushed via PHA. When the low byte is pushed on the stack, it's full, so the indexing wraps, putting that value at $01FF (the other end of the stack).

The indirect JMP instruction has a bug that's triggered if (and only if) the given address ends with $FF, as happens in this code. The CPU will construct the indirect address from the value at $0100 for the high byte and $01FF for the low byte (instead of from $01FF and $0200). But that separated address was precisely constructed with the aforementioned stack wrapping in mind, so it takes the program counter back to PT1, closing the infinite loop.

The indirect JMP bug was fixed in the later CMOS chips, breaking those rare bits of code that made use of it. One such example is the Apple II game *Randamn* (1983, Magnum Software). As part of its copy protection scheme, it writes a fake address to $02FF/$0300, and the other half of a true boot address to $0200. As such, it fails to boot on later Apple II machines that use the 65C02.

$30 Category: 6502

Code implements a left-shift 16-bit LFSR (linear feedback shift register). It visits every value in the set of values from 1 to 65535 in a deterministic shuffled-like order before repeating. This is the bit of mathematical magic behind graphical fizzle in/out effects as well as noise generators in sound chips. The Y register specifies the number of bits (1 to 16) to left shift into a 16-bit location.

As a left-shift LFSR, you can get away with having all your XOR tap points on just the low byte only. NES developer Brad Smith has generated all the single-byte left-shift polynomials for 8-bit, 16-bit, 24-bit, and 32-bit LFSRs. One of those polynomial bytes was used in this example.

A wider x86 LFSR is used in example 0x3b in *xchg rax,rax*.

$31 Category: 6502

Converts an 8-bit value to a 16-bit binary-coded decimal (BCD) value. Example based on code described by Andrew Jacobs, hosted on 6502.org.

BCD is often impractical for 6502 coding tasks. For simple decimal displays like game scores, most programmers will use a byte per digit rather than the nybble packing required by BCD.

Note: This code will not work on an NES, as Ricoh removed the (then-) patented BCD portion of the 6502 when they cloned it. Its D flag can still be managed with SED/CLD, but this doesn't change the CPU's ADC/SBC behavior.

$32 Category: Atari 2600

This method allows a BRK statement in Atari 2600 code to call a function and pass a byte parameter to it. When the function finishes, control is returned to the next statement following the BRK/parameter pair.

Thomas Jentzsch posted that he discovered this technique when disassembling the unreleased Atari 2600 game *The Lord of the Rings: Journey to Rivendell* (Parker Brothers, 1983). Since there was a function that needed to be called throughout the game code, programmer Mark Lesser used this BRK-call approach to save bytes.

When accessing RAM on the Atari 2600, address bit A8 is not decoded. This means that the 6502 stack (normally $0100 to $01FF) lives in page 0, along with the 128 bytes of RAM ($80 to $FF) and TIA graphics/sound registers ($00 to $2C).

This BRK parameter-passing implementation assumes a zero-page stack overlay. First, PLP discards the flags on the stack. Next, DEC $00,X changes the stack return address to point to the parameter that follows the BRK command, and LDA ($00,X) loads that parameter byte. Finally, RTS pops the address off the stack, increments it by one and sets the program counter. This resumes execution at the address immediately after the passed value.

Note: The 2600's crowded zero page has created other opportunities for some interesting cycle-saving tricks:

- *Combat* (Atari 2600 launch title, 1977) sets the missile 1 enable ($1E) and missile 0 enable ($1D) registers with two PHA commands, having first positioned the stack at $1E.
- In the 2600 *Pole Position* port (Atari, 1983), the stack pointer is positioned on the ball reset register ($14) and a BRK command is executed. This strobes (touching a register for effect, irrespective of data) the ball reset ($14), missile 1 reset ($13), and missile 0 reset ($12) registers using only three consecutive cycles. This trick is a key element of the cycle-critical road-drawing effect.

$33 Category: 6502

Eckhard Stolberg has popularized the "clockslide", a tunable cycle-exact delay. To tune the delay, additional code (not shown) performs an indexed jump to any chosen byte in the set of instructions.

Starting execution at the top of the slide consumes 15 cycles (the six CMP #$C9 instructions are two cycles each, and the CMP $EA is three cycles). Jumping to the top+1 results in the following instruction decoding: five CMP #$C9 instructions, a CMP #$C5, and a NOP, which is one fewer cycle (14 cycles). Top+2 is 13 cycles (five CMP #$C9 instructions + a CMP $EA), etc. So by choosing the appropriate slide entry point, every delay from 2 to 15 cycles is available (clockslides can, of course, be made arbitrarily longer).

See also Duff's Device, a technique credited to Lucasfilm developer Tom Duff in 1983.

$34 Category: 6502

Integer division of the accumulator by 3. Example from *Unsigned Integer Division Routines* (revision 2, June 21, 2014) by Omegamatrix, which has 31 accumulator-only (no X/Y usage) constant-cycle routines, one for each divisor of 2 through 32.

$35 Category: 6502

The 6502's instruction-decoding PLA creates many unintended (but sometimes viable) instructions. These are referred to as unintended, illegal, and/or pseudo instructions or ops. One such instruction is called "DCP" and will, for one of 7 addressing modes, perform a DEC followed by a CMP. The assembly ".BYTE $CF, <ADDR_1, >ADDR_1" could be written in some assemblers as "DCP ADDR_1".

This example decrements its way from $5A to $41 (which is "Z" to "A" in ASCII), stopping when the ADDR_1 value equals the accumulator value. Example adapted from *NMOS 6510 Unintended Opcodes - No More Secrets* by groepaz.

$36 Category: 6502

Computes the SIN of a byte, where the input degrees are given by the accumulator * 1.40625. Result is a signed 16-bit number, ranging over -/+0.99997. Example is from Hans Otten's recovered 6502 code examples from Lee Davidson.

$37 Category: 6502

Generates a CRC-32 over a given number of (256-byte) pages. The starting address of the data to be CRC-32ed must be page aligned, with the accumulator specifying the high byte of the start address. The X register specifies the number of pages to process.

CRCs are used to create short "fingerprints" of data. More pedantically, a CRC-32 computes the remainder of a polynomial division modulo 2 on an arbitrary-sized input, creating a (passably) uniformly-distributed 32-bit value.

This code produces the same output values as do the CRC-32s used by WinZip and 7-Zip (polynomial 0xEDB88320, working values starts with 0xFFFFFFFF, and final result is XORed with 0xFFFFFFFF as well). Code adapted from CRC-32 examples at nesdev.org.

$38 Category: 6502/65C02

The Z flag will be 0 if executed on an NMOS 6502, and 1 if executed on a CMOS 6502 (e.g. 65C02). This example from Lee Davidson's collection of very short code bits.

$39 Category: 6502

Computes the day of the week for any date in the year range 1900 to 2155. For X=month (1 to 12), A=day (1 to 31), and Y = year (1900+Y), it returns A = 1 (Sunday) to 7 (Saturday). Based on the concise code/logic designed by Paul Guertin (hosted on 6502.org).

The code computes the day of the week as (day + offset_table[month] + year + year/4 + adjust) mod 7. The offset changes the day count so the 1st of the month follows the last day of the previous month. The adjust is -1 after 2099 (as 2100 is not a leap year). The code begins by starting years in March to make leap years easier to compute. This is followed immediately by special handling for January and February in 1990. Later, PT3 computes (A-4) mod 7 (which is simpler to compute than A mod 7).

$3A Category: 6502

A generator for the solutions to the 8 queens problem. After each call, if the carry is clear, there is a valid solution in ZP_ADDR_1 to ZP_ADDR_1+7, where each byte value contains the column position of a queen that's in a row indexed by the byte's position. For example, the first solution is $07, $03, $00, $02, $05, $01, $06, $04. If the carry is set after the call, there are no more solutions (and the result is not a solution).

Here are the first three solutions found:

```
1st solution at | 2nd solution at | 3rd solution at
  ~80M cycles   |   ~96M cycles   |  ~105M cycles
   7----Q---    |    7---Q----    |    7-----Q--
   6------Q-    |    6------Q-    |    6---Q----
   5-Q------    |    5----Q---    |    5------Q-
   4-----Q--    |    4-Q------    |    4Q-------
   3--Q-----    |    3-----Q--    |    3--Q-----
   2Q-------    |    2Q-------    |    2----Q---
   1---Q----    |    1--Q-----    |    1-Q------
   0-------Q    |    0-------Q    |    0-------Q
    01234567    |     01234567    |     01234567
```

Using a non-recursive approach, queens are moved from the right to the left. Starting with the top row, when a queen finishes moving through a row, it resets, and the queen below it moves left one square. This sweeps 8^8 positions, with up to 7*(7+1)/2 = 28 queen pairings to compare per position. For shorter code, one can use the pairwise checks for both column and diagonal collisions (queens never leave their rows, so row collision checking is unnecessary). However, in this example, an 8-byte ZP_ADDR_2 structure was added, which quickly detects rook collisions with only a modest increase in code size. Only after determining that rook moves don't attack are the more costly pairwise diagonal checks performed.

$3B Category: 6502

Implementation of Bresenham's Line algorithm (developed in 1962), frequently used in 8-bit "wire-frame" games. (ADDR_1, ADDR_2) is the starting (X, Y) point, and (ADDR_3, ADDR_4) is the ending point. PTA initializes. PTB walks a step; if carry was set, the endpoint was reached.

Code was adapted from code posted by Petri Häkkinen to add endpoint checking.

$3C Category: 6502

The 81 bytes of data are compressed with a Lempel-Ziv (LZ77) variant. This code decompresses that data into 115 bytes, revealing Bilbo's birthday farewell quip "*I DON'T KNOW HALF OF YOU HALF AS WELL AS I SHOULD LIKE AND I LIKE LESS THAN HALF OF YOU HALF AS WELL AS YOU DESERVE*".

The compressor reads through input data, outputting character patterns it hasn't seen before (raw byte literals). For byte patterns found in the previous output, it outputs the offset and length of the match instead of the literal bytes. Many such LZ-based "crunchers" exist for 8-bit platforms. These days, the compressing phase of the workload is generally performed on modern machines, which may then wrap the results with an 8-bit-machine-specific decompressor.

The loop at PT2 outputs uncompressed literals (starting with "I DON'T KNOW HALF OF YOU") and the PT7 loop outputs referenced patterns (continuing with " HALF "). There is great variation in the algorithms that have evolved from LZ77, the 1977 algorithm designed by Lempel and Ziv. This code example uses TinyCrunch (v1.2, Christopher Jam, 2018), which minimizes the number of instructions needed to perform decompression. For size, TinyCrunch uses illegal opcodes and is two instructions shorter than what is shown in this example. Specifically, the LDA (ZP_ADDR16_2),Y / TAX pair is replaced by LAX(ZP_ADDR16_2),Y ($B3), and the SBC #$08 / TAX pair is replaced by SBX #8 ($CB).

$3D Category: 6502

Every byte has had its bit 5 flipped. If you XOR each byte with %00100000, then you get this runnable program:

```
AE 12 80   LDX $8012
AD 13 80   LDA $8013
5D EA 7F   EOR $7FEA,X
9D FE 7F   STA $7FFE,X
CA         DEX
E0 15      CPX #$15
D0 F2      BNE $8003
60         RTS
29 20      AND #%00100000
```

This new program should be stored at $8000 (as the comment suggests). When executed, the program flips bit 5 of each of its bytes...

```
8000   AE 12 80   LDX $8012
8003   AD 13 80   LDA $8013
8006   5D EA 7F   EOR $7FEA,X
8009   9D FE 7F   STA $7FFE,X
800C   CA         DEX
800D   E0 15      CPX #$15
800F   D0 F2      BNE $8003
8011   60         RTS
8012   29 20      AND #%00100000

8014   8E 32 A0   STX $A032
8017   8D 33 A0   STA $A033
801A   7D CA 5F   ADC $5FCA,X
801D   BD DE 5F   LDA $5FDE,X
8020   EA         NOP
8021   C0 35      CPY #$35
8023   F0 D2      BEQ $7FF7
8025   40         RTI
8026   09 00      ORA #$00
```

... and then appends the obfuscated code that you started with. :)

This puzzle is similar to a "quine", a term coined by Douglas Hofstadter (in his 1979 Pulitzer-Prize winning book *Gödel, Escher, Bach: an Eternal Golden Braid*) for a program that takes no input and produces a copy of its own code as output.

$3E Category: 6502

The code takes 14,598,366 cycles to execute, and 14,598,366 in hexadecimal is "DEC0DE". :)

Cycle counting was an important skill in the commercial Atari 2600 racing-the-beam days, and it's still essential for high-performance or timing-critical 8-bit coding. This puzzle is perhaps most easily solved by spreadsheet. If solving by emulator, the SEI is there so that interrupts don't add more cycles (though other sources of cycle stealing must be dealt with, such as a C64's VIC-II "bad lines", so Commodore readers should use a VIC-20 emulator instead).

Here's how each line contributes to the sum: 2 + 2 + 2 + 2 + 129030 + 5677320 + 8451465 + 129030 + 193292 + 506 + 506 + 6072 + 8855 + 1012 + 506 + 758 + 6.

The ORG $1000 is to show that no page crossing will occur (which would complicate cycle counting).

$3F Category: 6502

This program is written using only operators and operands that are 7-bit printable ASCII characters. This greatly limits the available instructions, values, and viable places in memory to host the code. Not available: any load, any store, in fact, any instruction that modifies RAM (including pseudo ops, so no self-modifying code). Also not available are any transfer instructions, any comparison instructions, and any ability to modify X or Y register values. The only branch instructions are BMI, BVC, and BVS, but only forward branching of 32 to 122. That said, it's possible to JMP anywhere by constructing values to PHA on the stack, then performing an RTI.

The program writes the string "BOLERO" to the stack. The string is constructed in an obfuscated way in a loop.

The disassembly:

```
         AND #$40
         AND #$3F
         SEC
         ADC #$41
PT1:     PHA
         SEC
         ADC #$7A
         ADC #$70
         ADC #$21
         PHA
         ADC #$7A
         ADC #$61
         ADC #$21
         PHA
         ADC #$32
         BMI PT2
         PLA
         PHA
         ADC #$7A
         ADC #$5D
         ADC #$21
         JMP PT1
```

"when you are a Bear of Very Little Brain, and you Think of Things, you find sometimes that a Thing which seemed very Thingish inside you is quite different when it gets out into the open and has other people looking at it" -- A. A. Milne, *The House at Pooh Corner* (1928)

Acknowledgments:

I'm blessed with an assembly-language dream team that edited, bug-fixed, and in multifold ways made this a better text. My astonishment and heartfelt gratitude goes out to:

– Kate Greenshields, for graciously reviewing an early draft

– Peter Ferrie, for astute technical feedback, and contributing Apple II anecdotes

– Michael Steil for sending me (legible) handwritten notes from his impressive 10-hour-flight playtesting session, and for rounding out examples with targeted Commodore minutiae

– Robin Harbron for contributing problem $23

– Jim Drew for suggesting problem $11

– And a special thank you to Nate Lawson, not just for his 6502 expertise and recommendations, but for providing much needed polish to all my prose.

Despite their efforts, there are undoubtedly a few errors still lurking in the text. When you find one, or more likely, if your OCD is going nuts because you know some detail that somehow eluded inclusion, then find me (Google will help) and let me know. I'll gather feedback at http://youdzone.com/8bit/errata.html

Appendix A: Instruction Reference

There are many 6502 instruction references online but usually not ordered by opcode byte value. This may be useful for evaluating some of the code examples in this book.

hex	binary	instruction	b	cyc	flags
00	00000000	BRK	1	7	I
01	00000001	ORA (addr,X)	2	6	N,Z
05	00000101	ORA zp	2	3	N,Z
06	00000110	ASL zp	2	5	N,Z,C
08	00001000	PHP	1	3	
09	00001001	ORA #imm	2	2	N,Z
0A	00001010	ASL	1	2	N,Z,C
0D	00001101	ORA addr	3	4	N,Z
0E	00001110	ASL addr	3	6	N,Z,C
10	00010000	BPL rel	2	2-4	
11	00010001	ORA (addr),Y	2	5-6	N,Z
15	00010101	ORA zp,X	2	4	N,Z
16	00010110	ASL zp,X	2	6	N,Z,C
18	00011000	CLC	1	2	C
19	00011001	ORA addr,Y	3	4-5	N,Z
1D	00011101	ORA addr,X	3	4-5	N,Z
1E	00011110	ASL addr,X	3	7	N,Z,C
20	00100000	JSR addr	3	6	
21	00100001	AND (addr,X)	2	6	N,Z
24	00100100	BIT zp	2	3	N,V,Z
25	00100101	AND zp	2	3	N,Z
26	00100110	ROL zp	2	5	N,Z,C
28	00101000	PLP	1	4	N,V,D,I,Z,C
29	00101001	AND #imm	2	2	N,Z
2A	00101010	ROL	1	2	N,Z,C
2C	00101100	BIT addr	3	4	N,V,Z
2D	00101101	AND addr	3	4	N,Z
2E	00101110	ROL addr	3	6	N,Z,C
30	00110000	BMI rel	2	2-4	
31	00110001	AND (addr),Y	2	5-6	N,Z
35	00110101	AND zp,X	2	4	N,Z
36	00110110	ROL zp,X	2	6	N,Z,C
38	00111000	SEC	1	2	C
39	00111001	AND addr,Y	3	4-5	N,Z
3D	00111101	AND addr,X	3	4-5	N,Z

hex	binary	instruction	b	cyc	flags
3E	00111110	ROL addr,X	3	7	N,Z,C
40	01000000	RTI	1	6	N,V,D,I,Z,C
41	01000001	EOR (addr,X)	2	6	N,Z
45	01000101	EOR zp	2	3	N,Z
46	01000110	LSR zp	2	5	N,Z,C
48	01001000	PHA	1	3	
49	01001001	EOR #imm	2	2	N,Z
4A	01001010	LSR	1	2	N,Z,C
4C	01001100	JMP addr	3	3	
4D	01001101	EOR addr	3	4	N,Z
4E	01001110	LSR addr	3	6	N,Z,C
50	01010000	BVC rel	2	2-4	
51	01010001	EOR (addr),Y	2	5-6	N,Z
55	01010101	EOR zp,X	2	4	N,Z
56	01010110	LSR zp,X	2	6	N,Z,C
58	01011000	CLI	1	2	I
59	01011001	EOR addr,Y	3	4-5	N,Z
5D	01011101	EOR addr,X	3	4-5	N,Z
5E	01011110	LSR addr,X	3	7	N,Z,C
60	01100000	RTS	1	6	
61	01100001	ADC (addr,X)	2	6	N,V,Z,C
65	01100101	ADC zp	2	3	N,V,Z,C
66	01100110	ROR zp	2	5	N,Z,C
68	01101000	PLA	1	4	N,Z
69	01101001	ADC #imm	2	2	N,V,Z,C
6A	01101010	ROR	1	2	N,Z,C
6C	01101100	JMP (addr)	3	5	
6D	01101101	ADC addr	3	4	N,V,Z,C
6E	01101110	ROR addr	3	6	N,Z,C
70	01110000	BVS rel	2	2-4	
71	01110001	ADC (addr),Y	2	5-6	N,V,Z,C
75	01110101	ADC zp,X	2	4	N,V,Z,C
76	01110110	ROR zp,X	2	6	N,Z,C
78	01111000	SEI	1	2	I
79	01111001	ADC addr,Y	3	4-5	N,V,Z,C
7D	01111101	ADC addr,X	3	4-5	N,V,Z,C
7E	01111110	ROR addr,X	3	7	N,Z,C
81	10000001	STA (addr,X)	2	6	
84	10000100	STY zp	2	3	
85	10000101	STA zp	2	3	
86	10000110	STX zp	2	3	
88	10001000	DEY	1	2	N,Z
8A	10001010	TXA	1	2	N,Z
8C	10001100	STY addr	3	4	
8D	10001101	STA addr	3	4	
8E	10001110	STX addr	3	4	

hex	binary	instruction	b	cyc	flags
90	10010000	BCC rel	2	2-4	
91	10010001	STA (addr),Y	2	6	
94	10010100	STY zp,X	2	4	
95	10010101	STA zp,X	2	4	
96	10010110	STX zp,Y	2	4	
98	10011000	TYA	1	2	N,Z
99	10011001	STA addr,Y	3	5	
9A	10011010	TXS	1	2	
9D	10011101	STA addr,X	3	5	
A0	10100000	LDY #imm	2	2	N,Z
A1	10100001	LDA (addr,X)	2	6	N,Z
A2	10100010	LDX #imm	2	2	N,Z
A4	10100100	LDY zp	2	3	N,Z
A5	10100101	LDA zp	2	3	N,Z
A6	10100110	LDX zp	2	3	N,Z
A8	10101000	TAY	1	2	N,Z
A9	10101001	LDA #imm	2	2	N,Z
AA	10101010	TAX	1	2	N,Z
AC	10101100	LDY addr	3	4	N,Z
AD	10101101	LDA addr	3	4	N,Z
AE	10101110	LDX addr	3	4	N,Z
B0	10110000	BCS rel	2	2-4	
B1	10110001	LDA (addr),Y	2	5-6	N,Z
B4	10110100	LDY zp,X	2	4	N,Z
B5	10110101	LDA zp,X	2	4	N,Z
B6	10110110	LDX zp,Y	2	4	N,Z
B8	10111000	CLV	1	2	V
B9	10111001	LDA addr,Y	3	4-5	N,Z
BA	10111010	TSX	1	2	N,Z
BC	10111100	LDY addr,X	3	4-5	N,Z
BD	10111101	LDA addr,X	3	4-5	N,Z
BE	10111110	LDX addr,Y	3	4-5	N,Z
C0	11000000	CPY #imm	2	2	N,Z,C
C1	11000001	CMP (addr,X)	2	6	N,Z,C
C4	11000100	CPY zp	2	3	N,Z,C
C5	11000101	CMP zp	2	3	N,Z,C
C6	11000110	DEC zp	2	5	N,Z
C8	11001000	INY	1	2	N,Z
C9	11001001	CMP #imm	2	2	N,Z,C
CA	11001010	DEX	1	2	N,Z
CC	11001100	CPY addr	3	4	N,Z,C
CD	11001101	CMP addr	3	4	N,Z,C
CE	11001110	DEC addr	3	6	N,Z
D0	11010000	BNE rel	2	2-4	
D1	11010001	CMP (addr),Y	2	5-6	N,Z,C
D5	11010101	CMP zp,X	2	4	N,Z,C

hex	binary	instruction	b	cyc	flags
D6	11010110	DEC zp,X	2	6	N,Z
D8	11011000	CLD	1	2	D
D9	11011001	CMP addr,Y	3	4-5	N,Z,C
DD	11011101	CMP addr,X	3	4-5	N,Z,C
DE	11011110	DEC addr,X	3	7	N,Z
E0	11100000	CPX #imm	2	2	N,Z,C
E1	11100001	SBC (addr,X)	2	6	N,V,Z,C
E4	11100100	CPX zp	2	3	N,Z,C
E5	11100101	SBC zp	2	3	N,V,Z,C
E6	11100110	INC zp	2	5	N,Z
E8	11101000	INX	1	2	N,Z
E9	11101001	SBC #imm	2	2	N,V,Z,C
EA	11101010	NOP	1	2	
EC	11101100	CPX addr	3	4	N,Z,C
ED	11101101	SBC addr	3	4	N,V,Z,C
EE	11101110	INC addr	3	6	N,Z
F0	11110000	BEQ rel	2	2-4	
F1	11110001	SBC (addr),Y	2	5-6	N,V,Z,C
F5	11110101	SBC zp,X	2	4	N,V,Z,C
F6	11110110	INC zp,X	2	6	N,Z
F8	11111000	SED	1	2	D
F9	11111001	SBC addr,Y	3	4-5	N,V,Z,C
FD	11111101	SBC addr,X	3	4-5	N,V,Z,C
FE	11111110	INC addr,X	3	7	N,Z

Appendix B: Further Reading

xorpd. 2014. *xchg rax,rax*. North Charleston, SC: Createspace Independent Publishing Platform.

Warren, Henry S. 2012. *Hacker's Delight*. 2nd ed. Boston, MA: Addison-Wesley Educational.

groepaz. 2022. *NMOS 6510 Unintended Opcodes*. Version 0.97. https://csdb.dk/getinternalfile.php/239398/NoMoreSecrets-NMOS6510UnintendedOpcodes-20222412.pdf

Butterfield, Jim. 1984. *Machine Language for the Commodore 64 and Other Commodore Computers*. Bowie, MD: Brady Communications Company, Inc.

Wagner, Roger. 2014. *Assembly Lines: The Complete Book*. Morrisville, NC: Lulu.com.

Toledo Gutierrez, Oscar. 2022. *Programming Games for Atari 2600*. Morrisville, NC: Lulu.com.

Colophon

Book text was rendered in Charis SIL, a transitional serif typeface developed by SIL International and licensed under the SIL Open Font License (OFL).

Code segments were rendered in Liberation Mono, part of the Liberation font family originally developed by Ascender Corporation. As of December 2018 it is licensed under OFL.

Cover text rendered in Merriweather, designed by Eben Sorkin, and is also licensed under OFL.

My intent is to earn $0 from this book. The e-book should be free, and the paperback should be as near to printing costs as possible. Any unavoidable earnings will be donated to the Sempervirens Fund, a land trust set up to preserve coast redwoods in California's Santa Cruz Mountains.