ZERO PAGE

OPERATOR REFERENCE MANUAL

Zero Page Initiative -- Field Operations Division

2026

CONTENTS

ZERO PAGE

DOCUMENT CLASS: OPERATOR REFERENCE MANUAL
DEVICE: ZERO PAGE MICROCOMPUTER
ARCHITECTURE: MOS 6502 COMPATIBLE
INSTRUCTION SET: ZERO PAGE SUBSET (AUTHORISED MNEMONICS)

CLASSIFICATION: FIELD ISSUE // CONTRACTOR USE ONLY
MANUAL REV: 1.0
BUILD VERSION: 0.1.0
ISSUED BY: ZERO PAGE INITIATIVE
FIELD OPERATIONS DIVISION

PROPERTY OF: ZERO PAGE INITIATIVE
SYSTEMS CONTRACTING GROUP
AUTHORISED OPERATOR: [REDACTED]

NOTICE TO OPERATOR

This manual describes the operational characteristics of the Zero Page Microcomputer as issued to field contractors. Deviations from documented behaviour must be reported through your assigned channel. Unauthorised modifications to system memory are logged.

Retain this manual for the duration of your assignment. Return is not required. Destruction is not required.

However, this manual should not exist outside authorised contractor facilities.

Zero Page Initiative assumes no liability for outcomes resulting from improper instruction use.

DEVICE ORIENTATION

```
┌─ SECTION 1 — DEVICE ORIENTATION ──────────────────────┐
│  ZERO PAGE MICROCOMPUTER                              │
│  CONTRACTOR ORIENTATION GUIDE                         │
└───────────────────────────────────────────────────────┘
```

WHAT YOU ARE OPERATING

The Zero Page Microcomputer is a 6502-class single-board unit deployed for field data-processing tasks. You have been contracted to write programs in 6502 assembly that transform input values stored in the zero page ($00--$FF) into required output values at designated output addresses.

Each assignment is a specification. The specification defines:

- which memory addresses contain input values
- which memory addresses must hold output values when the program halts
- the cycle budget (how many instructions may execute)
- what BRK means (your program must reach BRK to pass)

The machine does not distinguish between a contractor who understands the problem and one who does not. It runs the program and checks memory. The result is either correct or it is not.

OPERATOR RESPONSIBILITIES

You write programs. The machine runs them. The tests verify the output. If the tests pass, you are cleared for the next assignment.

You are not required to understand the purpose of each

computation. Field contractors are issued assignments; the rationale is maintained at a higher classification level. Your responsibility is correctness within the specified I/O contract and cycle limit.

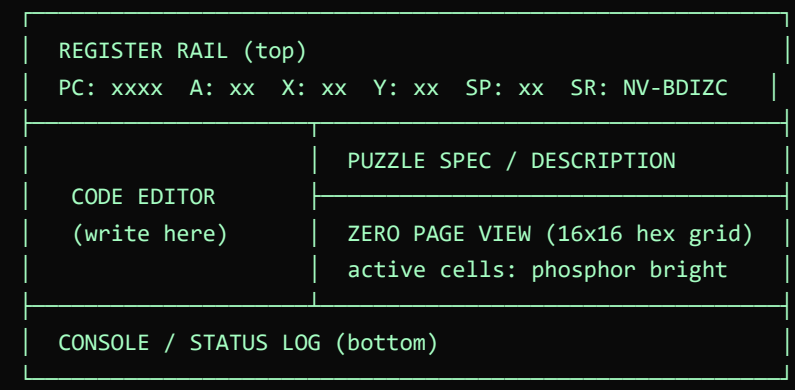
DEVICE HISTORY

The Zero Page Microcomputer runs a 6502-class architecture. MOS Technology released the original 6502 in 1975 as an 8-bit processor. The chip went on to power the Apple II, the Commodore 64, the BBC Micro, the Atari 2600, and the NES. The architecture documented here is a subset appropriate for zero-page-constrained data processing.

Memory above the zero page is available for the stack ($0100--$01FF) and for scratch space as needed.

The simulator honours standard 6502 flag semantics with one documented exception: Decimal mode (D flag / SED instruction) is not implemented. BCD arithmetic will not produce correct results. Do not use SED in production programs.

TERMINAL LAYOUT



The code editor is where you write programs. The zero-page view shows live memory state as your program executes. The register rail shows CPU state. The console reports run results, test verdicts, and error messages.

PROGRAM STRUCTURE

A minimal program:

```
LDA $00     ; load input from address $00
STA $02     ; store to output address $02
BRK         ; halt -- required to pass tests
```

Programs execute from address $00 in the program counter (the simulator maps your code to execution memory). Comments begin with ; and are stripped before assembly.

Labels are supported:

```
LOOP:
    LDA $00,X
    STA $10,X
    INX
    CPX #$08
    BNE LOOP
    BRK
```

Numeric literals: #$nn is immediate hex, #nn is immediate decimal. Addresses: $nn is zero-page hex, $nnnn is absolute hex.

NOTATION USED IN THIS MANUAL

```
$nn        -- 8-bit hex address or value (zero page)
$nnnn      -- 16-bit hex address (absolute)
#$nn       -- immediate literal value
A          -- accumulator register
X          -- X index register
Y          -- Y index register
SP         -- stack pointer (initialised to $FF)
PC         -- program counter
SR         -- status register (NV-BDIZC flags)
[addr]     -- value at memory address addr
M          -- memory operand (context-dependent)
```

Flag abbreviations used throughout:

```
N  -- Negative flag  (bit 7 of SR)
V  -- oVerflow flag  (bit 6)
-  -- unused         (bit 5, always 1)
B  -- Break flag     (bit 4)
D  -- Decimal flag   (bit 3, not implemented)
I  -- Interrupt flag (bit 2)
Z  -- Zero flag      (bit 1)
C  -- Carry flag     (bit 0)
```

SAVING AND LOADING

Programs are saved as .6502 files containing your code and current progress state. The save mechanism is exposed through the terminal interface. Progress is per-assignment.

Best cycle scores are retained per puzzle. The machine reports your score against the cycle limit on each run.

INSTRUCTION REFERENCE

```
┌─ SECTION 2 — 6502 INSTRUCTION REFERENCE ──────────────────┐
│  AUTHORISED MNEMONIC SET -- ZERO PAGE MICROCOMPUTER       │
│  ZERO PAGE SUBSET + STACK + SUBROUTINES + INDIRECT MODES   │
└────────────────────────────────────────────────────────────┘
```

ADDRESSING MODES

Each instruction operates in one or more addressing modes. The mode determines how the operand is interpreted.

Mode	Syntax	Description
Immediate	#$nn	Literal value in instruction
Zero-page	$nn	Address in range $00--$FF
Zero-page,X	$nn,X	Zero-page addr + X (wraps at $FF)
Zero-page,Y	$nn,Y	Zero-page addr + Y (wraps at $FF)
Absolute	$nnnn	16-bit address
Absolute,X	$nnnn,X	16-bit address + X
Absolute,Y	$nnnn,Y	16-bit address + Y
Indirect	($nnnn)	JMP only: jump to address at $nnnn
(Indirect,X)	($nn,X)	Add X to $nn, read 16-bit ptr there
(Indirect),Y	($nn),Y	Read 16-bit ptr at $nn, add Y
Implied	(no operand)	Register or stack operation

Indirect addressing notes. For ($nn,X): add X to the zero-page address, read a 16-bit little-endian pointer from that location, use the pointer as the effective address. For ($nn),Y: read a 16-bit pointer from $nn/$nn+1, add Y to get the effective address. Both modes require the base zero-page address ($nn) to be less than $FE to avoid wrap.

LOAD / STORE

LDA — LOAD ACCUMULATOR

Syntax: LDA operand
Loads the operand value into register A.
Flags: N, Z

LDA #$nn	immediate
LDA $nn	zero-page
LDA $nn,X	zero-page indexed X
LDA $nn,Y	zero-page indexed Y
LDA $nnnn	absolute
LDA $nnnn,X	absolute indexed X
LDA $nnnn,Y	absolute indexed Y
LDA ($nn,X)	indexed indirect
LDA ($nn),Y	indirect indexed

LDX — LOAD X REGISTER

Syntax: LDX operand
Loads the operand value into register X.
Flags: N, Z

LDX #$nn	immediate
LDX $nn	zero-page
LDX $nn,Y	zero-page indexed Y
LDX $nnnn	absolute

LDY — LOAD Y REGISTER

Syntax: LDY operand
Loads the operand value into register Y.
Flags: N, Z

```
|  LDY #$nn       immediate                          |
|  LDY $nn        zero-page                          |
|  LDY $nn,X      zero-page indexed X                |
|  LDY $nnnn      absolute                           |
└────────────────────────────────────────────────────┘

┌────────────────────────────────────────────────────┐
|  STA  STORE ACCUMULATOR                            |
├────────────────────────────────────────────────────┤
|  Syntax: STA addr                                  |
|  Stores the value of A into memory. Does not alter A.  |
|  Flags: none                                       |
|                                                    |
|  STA $nn        zero-page                          |
|  STA $nn,X      zero-page indexed X                |
|  STA $nnnn      absolute                           |
|  STA $nnnn,X    absolute indexed X                 |
|  STA $nnnn,Y    absolute indexed Y                 |
|  STA ($nn,X)    indexed indirect                   |
|  STA ($nn),Y    indirect indexed                   |
└────────────────────────────────────────────────────┘

┌────────────────────────────────────────────────────┐
|  STX  STORE X REGISTER                             |
├────────────────────────────────────────────────────┤
|  Syntax: STX addr                                  |
|  Stores X into memory. Does not alter X.           |
|  Flags: none                                       |
|                                                    |
|  STX $nn        zero-page                          |
|  STX $nn,Y      zero-page indexed Y                |
|  STX $nnnn      absolute                           |
└────────────────────────────────────────────────────┘

┌────────────────────────────────────────────────────┐
|  STY  STORE Y REGISTER                             |
├────────────────────────────────────────────────────┤
|  Syntax: STY addr                                  |
|  Stores Y into memory. Does not alter Y.           |
```

```
|  Flags: none                                          |
|                                                       |
|  STY $nn        zero-page                             |
|  STY $nn,X      zero-page indexed X                   |
|  STY $nnnn      absolute                              |
└───────────────────────────────────────────────────────┘
```

REGISTER TRANSFERS

```
┌───────────────────────────────────────────────────────┐
|  TAX  TRANSFER A TO X     Flags: N, Z                 |
|  TAY  TRANSFER A TO Y     Flags: N, Z                 |
|  TXA  TRANSFER X TO A     Flags: N, Z                 |
|  TYA  TRANSFER Y TO A     Flags: N, Z                 |
|  TSX  TRANSFER SP TO X    Flags: N, Z                 |
|  TXS  TRANSFER X TO SP    Flags: none                 |
└───────────────────────────────────────────────────────┘
```

All are implied-mode (no operand). Each copies the source register to the destination. TXS does not set flags (SP is not a data register).

STACK OPERATIONS

The stack occupies $0100--$01FF. SP points to the next free slot (pre-decrement push / post-increment pop). SP is initialised to $FF at reset. Stack grows downward.

```
┌───────────────────────────────────────────────────────┐
|  PHA  PUSH ACCUMULATOR                                |
|  Pushes A onto the stack. SP -= 1. Flags: none.       |
|                                                       |
|  PLA  PULL ACCUMULATOR                                |
|  Pulls top of stack into A. SP += 1. Flags: N, Z.     |
|                                                       |
|  PHP  PUSH PROCESSOR STATUS                           |
|  Pushes SR onto the stack. SP -= 1. Flags: none.      |
|  (B flag is set in the pushed byte.)                  |
```

```
┌──────────────────────────────────────────────────────┐
│                                                      │
│  PLP  PULL PROCESSOR STATUS                          │
│  Pulls top of stack into SR. SP += 1. Flags: all.    │
└──────────────────────────────────────────────────────┘
```

Common use: preserve A across a subroutine that modifies it.

```
    PHA             ; save A
    JSR SUBROUTINE
    PLA             ; restore A
```

ARITHMETIC

Carry flag is required for ADC and SBC. Always set carry state explicitly before performing addition or subtraction.

```
┌──────────────────────────────────────────────────────┐
│  CLC  CLEAR CARRY      -- use before ADC             │
│  SEC  SET CARRY        -- use before SBC             │
└──────────────────────────────────────────────────────┘
```

```
┌──────────────────────────────────────────────────────┐
│  ADC  ADD WITH CARRY                                 │
├──────────────────────────────────────────────────────┤
│  A = A + M + C                                       │
│  Flags: N, V, Z, C                                   │
│  Always precede with CLC unless chaining multi-byte add. │
│                                                      │
│  ADC #$nn       immediate                            │
│  ADC $nn        zero-page                            │
│  ADC $nn,X      zero-page indexed X                  │
│  ADC $nnnn      absolute                             │
└──────────────────────────────────────────────────────┘
```

```
┌──────────────────────────────────────────────────────┐
│  SBC  SUBTRACT WITH CARRY (BORROW)                   │
├──────────────────────────────────────────────────────┤
│  A = A - M - (1 - C)                                 │
│  Flags: N, V, Z, C                                   │
│  Always precede with SEC (sets borrow-not to 0).     │
```

```
|                                                          |
|  SBC #$nn       immediate                                |
|  SBC $nn        zero-page                                |
|  SBC $nn,X      zero-page indexed X                      |
|  SBC $nnnn      absolute                                 |
└──────────────────────────────────────────────────────────┘
```

V (overflow) flag is set when the signed result exceeds the 8-bit signed range (-128..127). C (carry) is set when the unsigned result exceeds 255 (ADC) or when no borrow occurred (SBC -- inverted carry semantics for subtract).

Multi-byte addition. Add the low bytes with CLC/ADC, then add the high bytes with ADC (no CLC -- carry propagates).

LOGIC

```
┌──────────────────────────────────────────────────────────┐
│  AND  BITWISE AND                                        │
│  A = A AND M    Flags: N, Z                              │
│  Use to mask off (zero) specific bits.                   │
│  Example: AND #$0F isolates the low nibble.              │
│                                                          │
│  AND #$nn  AND $nn  AND $nn,X  AND $nnnn                 │
└──────────────────────────────────────────────────────────┘

┌──────────────────────────────────────────────────────────┐
│  ORA  BITWISE OR                                         │
│  A = A OR M     Flags: N, Z                              │
│  Use to set specific bits.                               │
│  Example: ORA #$80 sets bit 7.                           │
│                                                          │
│  ORA #$nn  ORA $nn  ORA $nn,X  ORA $nnnn                 │
└──────────────────────────────────────────────────────────┘

┌──────────────────────────────────────────────────────────┐
│  EOR  BITWISE EXCLUSIVE OR                               │
│  A = A XOR M    Flags: N, Z                              │
│  Use to toggle specific bits, or as a checksum.          │
```

```
|  Example: EOR #$FF inverts all bits.                    |
|                                                         |
|  EOR #$nn  EOR $nn  EOR $nn,X  EOR $nnnn                |
└─────────────────────────────────────────────────────────┘
```

SHIFT AND ROTATE

All shifts operate on either A (accumulator) or a memory byte. All affect N, Z, and C flags. Rotate instructions include the carry bit in the rotation.

```
┌─────────────────────────────────────────────────────────┐
|  ASL  ARITHMETIC SHIFT LEFT                             |
|  bit 7 -> C ; bits shifted left ; bit 0 = 0             |
|  Equivalent to multiply by 2 (unsigned, if C=0 after).  |
|  Flags: N, Z, C                                         |
|                                                         |
|  ASL A    (accumulator)                                 |
|  ASL $nn  (zero-page)                                   |
└─────────────────────────────────────────────────────────┘
```

```
┌─────────────────────────────────────────────────────────┐
|  LSR  LOGICAL SHIFT RIGHT                               |
|  bit 0 -> C ; bits shifted right ; bit 7 = 0            |
|  Equivalent to unsigned divide by 2.                    |
|  Flags: N=0, Z, C                                       |
|                                                         |
|  LSR A    (accumulator)                                 |
|  LSR $nn  (zero-page)                                   |
└─────────────────────────────────────────────────────────┘
```

```
┌─────────────────────────────────────────────────────────┐
|  ROL  ROTATE LEFT THROUGH CARRY                         |
|  old C -> bit 0 ; bit 7 -> new C ; all other bits left  |
|  Flags: N, Z, C                                         |
|  State of C BEFORE ROL becomes bit 0 of result.         |
|  CLC before ROL if you want a clean left shift.         |
|                                                         |
|  ROL A    (accumulator)                                 |
```

```
|  ROL $nn  (zero-page)                                          |
└────────────────────────────────────────────────────────────────┘
```

```
┌────────────────────────────────────────────────────────────────┐
|  ROR  ROTATE RIGHT THROUGH CARRY                               |
|  old C -> bit 7 ; bit 0 -> new C ; all other bits right        |
|  Flags: N, Z, C                                                |
|  State of C BEFORE ROR becomes bit 7 of result.                |
|                                                                |
|  ROR A    (accumulator)                                        |
|  ROR $nn  (zero-page)                                          |
└────────────────────────────────────────────────────────────────┘
```

COMPARE

Compare instructions perform a subtraction but discard the result. They set flags without modifying A, X, or Y.

```
┌────────────────────────────────────────────────────────────────┐
|  CMP  COMPARE ACCUMULATOR                                      |
|  Sets N, Z, C based on (A - M).                                |
|  A is NOT modified.                                            |
|                                                                |
|  Result interpretation:                                        |
|    A = M : Z=1, C=1, N=0                                       |
|    A > M : Z=0, C=1, N=0 (unsigned)                            |
|    A < M : Z=0, C=0, N=1 (unsigned)                            |
|                                                                |
|  CMP #$nn  CMP $nn  CMP $nn,X  CMP $nnnn                       |
└────────────────────────────────────────────────────────────────┘
```

```
┌────────────────────────────────────────────────────────────────┐
|  CPX  COMPARE X REGISTER                                       |
|  Sets N, Z, C based on (X - M).                                |
|  X is NOT modified.                                            |
|                                                                |
|  CPX #$nn  CPX $nn  CPX $nnnn                                  |
└────────────────────────────────────────────────────────────────┘
```

```
┌──────────────────────────────────────────────────────┐
│  CPY  COMPARE Y REGISTER                             │
│  Sets N, Z, C based on (Y - M).                      │
│  Y is NOT modified.                                  │
│                                                      │
│  CPY #$nn  CPY $nn  CPY $nnnn                        │
└──────────────────────────────────────────────────────┘
```

BRANCH INSTRUCTIONS

Branch instructions jump to a relative offset if the named condition is true. Offset range: -128 to +127 bytes from the instruction following the branch.

```
┌──────────────────────────────────────────────────────┐
│  Mnemonic  Condition       Flag test                 │
│  --------  ---------       ---------                 │
│  BEQ       Branch if Equal  Z = 1                     │
│  BNE       Branch if Not=   Z = 0                     │
│  BCS       Branch if C Set  C = 1  (unsigned >=)      │
│  BCC       Branch if C Clr  C = 0  (unsigned <)       │
│  BMI       Branch if Minus  N = 1  (signed negative)  │
│  BPL       Branch if Plus   N = 0  (signed non-negative)
│  BVS       Branch if V Set  V = 1  (signed overflow)  │
│  BVC       Branch if V Clr  V = 0  (no signed overflow)
└──────────────────────────────────────────────────────┘
```

Branch targets are written as labels in source:

```
    CPX #$08
    BNE LOOP      ; branch back to LOOP if X != 8
```

JUMP AND SUBROUTINE

```
┌──────────────────────────────────────────────────────┐
│  JMP  JUMP                                           │
│  Sets PC to the target address. Unconditional.       │
│                                                      │
│  JMP $nnnn     absolute jump                         │
```

```
|  JMP ($nnnn)   indirect jump (reads target from $nnnn)   |
```

```
|  JSR  JUMP TO SUBROUTINE                                 |
|  Pushes PC-1 (return address) onto stack, then jumps.    |
|  Stack usage: 2 bytes.                                   |
|                                                          |
|  JSR $nnnn     absolute (label in practice)              |
```

```
|  RTS  RETURN FROM SUBROUTINE                             |
|  Pulls return address from stack, sets PC to addr+1.     |
|  Every JSR must be balanced by exactly one RTS.          |
```

Subroutine pattern:

```
    JSR MYSUB       ; call
    ...             ; continues here after RTS
    BRK

MYSUB:
    ; body
    RTS             ; return
```

Subroutines may nest. Each nesting level consumes 2 bytes of stack. Stack is 256 bytes; avoid deep recursion.

INCREMENT / DECREMENT

```
|  INC  INCREMENT MEMORY     M = M + 1   Flags: N, Z       |
|  DEC  DECREMENT MEMORY     M = M - 1   Flags: N, Z       |
|                                                          |
|  INC $nn  INC $nn,X  INC $nnnn                           |
|  DEC $nn  DEC $nn,X  DEC $nnnn                           |
```

INX	INCREMENT X	X = X + 1	Flags: N, Z
DEX	DECREMENT X	X = X - 1	Flags: N, Z
INY	INCREMENT Y	Y = Y + 1	Flags: N, Z
DEY	DECREMENT Y	Y = Y - 1	Flags: N, Z

(all implied mode, no operand)

INC and DEC wrap: $00 - 1 = $FF; $FF + 1 = $00. INX/DEX/INY/DEY wrap the same way. These do NOT affect the carry flag.

STATUS FLAG INSTRUCTIONS

CLC	Clear Carry flag	C = 0	
SEC	Set Carry flag	C = 1	
CLD	Clear Decimal flag	D = 0	(no-op here)
SED	Set Decimal flag	D = 1	(not implemented)
CLI	Clear Interrupt flag	I = 0	
SEI	Set Interrupt flag	I = 1	
CLV	Clear Overflow flag	V = 0	

Note: SED is listed for completeness. Decimal mode is not implemented in this simulator. BCD arithmetic will not produce correct results. Use SED only if you have a specific reason and understand the behaviour.

SYSTEM

BRK BREAK / HALT

Halts execution. All programs must terminate with BRK. Tests are not evaluated until BRK is reached. Sets B flag in SR before halting.

```
┃  NOP  NO OPERATION                                          ┃
┃  Consumes one cycle. Does nothing.                          ┃
┃  Flags: none.                                               ┃
┃  Use when you need to consume a cycle or pad code.          ┃
┗━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┛
```

STATUS REGISTER REFERENCE

The Status Register (SR) is an 8-bit register. Bit layout:

```
  Bit:  7   6   5   4   3   2   1   0
  Flag: N   V   -   B   D   I   Z   C
        |   |   |   |   |   |   |   |
        |   |   |   |   |   |   |   +-- Carry
        |   |   |   |   |   |   +------ Zero
        |   |   |   |   |   +---------- Interrupt disable
        |   |   |   |   +-------------- Decimal (not impl.)
        |   |   |   +------------------ Break
        |   |   +---------------------- Unused (always 1)
        |   +-------------------------- Overflow
        +------------------------------ Negative
```

N (Negative): Set when bit 7 of the result is 1. Useful for detecting negative signed values or checking bit 7.

V (Overflow): Set when a signed arithmetic result overflows the -128..127 range. E.g.: $7F + $01 = $80 (sets V because 127 + 1 = -128 in signed interpretation).

Z (Zero): Set when the result is exactly $00.

C (Carry): For addition: set when result > $FF (unsigned overflow). For subtraction: set when result >= $00 (no borrow). For shifts: the bit shifted out.

MEMORY MAP

```
┌─ SECTION 3 — MEMORY MAP AND I/O REFERENCE ──────────────┐
│  ZERO PAGE MICROCOMPUTER                                │
│  ADDRESS SPACE: $0000 -- $FFFF                          │
└─────────────────────────────────────────────────────────┘
```

ZERO PAGE -- $0000 TO $00FF

The zero page is the primary data region for all mission programs. I/O contracts are defined exclusively within this range. All 6502 zero-page addressing modes operate here with 1-byte addresses.

```
$00        -- Puzzle input / general purpose
$01        -- Puzzle input / general purpose
$02        -- Puzzle output / general purpose
  ...
$FF        -- top of zero page (special use in sandbox; see below)
```

During mission execution, the test harness seeds specific addresses with input values before your program runs, then reads output addresses after BRK. Each puzzle specifies exactly which addresses are inputs and which are outputs. Addresses not specified by the puzzle are available as scratch space.

Zero-page scratch: Any address not listed in the puzzle's I/O contract is available for temporary storage. Using $00 as a scratch register when $00 is an input address will corrupt the test -- read the I/O spec for each puzzle.

STACK -- $0100 TO $01FF

The processor stack. Used by PHA/PLA, PHP/PLP, and JSR/RTS. SP is initialised to $FF at reset, so the first push lands at $01FF and SP becomes $FE.

The stack cannot be directly addressed by LDA/STA in this simulator -- use the stack only through the stack instructions (PHA, PLA, PHP, PLP, JSR, RTS).

Stack overflow (SP underflows below $00) and underflow (SP increments above $FF) are not explicitly trapped. Keep stack depth proportional to your program's needs.

MISSION PROGRAMS -- $0200 ONWARD

Assembled program bytes are mapped to a separate code region beginning at $0200. PC is initialised to $0200 at run start. This region does not overlap with zero-page data.

MEMORY MAP SUMMARY

Address Range	Region	Notes
$0000--$00FF	Zero Page	Mission I/O + scratch
$0100--$01FF	Stack	SP-managed, PHA/JSR
$0200+	Program	PC starts at $0200
$FE	(Sandbox only)	RNG register
$FF	(Sandbox only)	Keyboard input

SANDBOX-SPECIFIC I/O -- $FE AND $FF

Sandbox mode is an ungraded free-execution environment. Two memory addresses have special behaviour in sandbox mode only. These addresses are NOT active during mission programs.

```
┌──────────────────────────────────────────────────────┐
│  $FE  -- RNG REGISTER                                │
│  Read-only. Each LDA $FE generates a fresh pseudo-random │
│  byte (0--255) and loads it into A. The value is not     │
│  stored in memory -- each read produces a new number.    │
│  Write to $FE has no special effect.                     │
│                                                      │
│  Example:                                            │
│    LDA $FE     ; A = random byte 0--255              │
│    STA $02     ; store it                            │
└──────────────────────────────────────────────────────┘
```

```
┌──────────────────────────────────────────────────────┐
│  $FF  -- KEYBOARD INPUT REGISTER                     │
│  Read-only. Holds the last key pressed as a byte.        │
│  Cleared to $00 after each LDA $FF (auto-consume).       │
│  Arrow key codes: Up=$01 Down=$02 Left=$03 Right=$04     │
│  Printable keys: ASCII value                         │
│                                                      │
│  Example:                                            │
│    LDA $FF     ; A = last keypress (or $00 if none)  │
│    STA $02     ; store for processing                │
└──────────────────────────────────────────────────────┘
```

FRAMEBUFFER -- $80 TO $FD (SANDBOX)

Sandbox programs that generate visual output use the framebuffer region at $80--$FD. This is a 16x8 grid (128 cells, 1 bit per cell for presence). Cell address: cell = y * 16 + x, stored at $80 + cell.

The framebuffer is a sandbox feature. Mission programs do not use it.

INTERRUPT AND RESET VECTORS

Standard 6502 vector locations are defined in ROM:

```
$FFFA/$FFFB  -- NMI vector     (not used in this simulator)
$FFFC/$FFFD  -- RESET vector   (initialises PC to $0200)
$FFFE/$FFFF  -- IRQ/BRK vector (not used in this simulator)
```

Hardware interrupts are not implemented. BRK halts execution directly without invoking an interrupt handler.

MISSION SPECIFICATIONS

```
┌─ SECTION 4 — MISSION SPECIFICATIONS ──────────────────────┐
│  ZERO PAGE INITIATIVE -- FIELD PROCESSING ASSIGNMENTS     │
│  CLUSTER 0 THROUGH CLUSTER 3                              │
│  24 MISSIONS TOTAL -- 15 MAIN / 6 BRIDGE / 3 HIDDEN       │
└───────────────────────────────────────────────────────────┘
```

Each entry on the following pages is a mission specification sheet. Read the I/O RANGE and CYCLE LIMIT before writing any code. The WORKING KNOWLEDGE block lists facts about the instruction set that are relevant to this puzzle -- not hints toward a specific solution.

NEW INSTRUCTIONS marks mnemonics introduced for the first time at this point in the campaign.

```
┌─ P1 — FIELD BALLISTICS UNIT -- PROTO A ─── CLUSTER 0 ──────┐
│  CLASSIFICATION: TRAINING // AUTHORISED DISTRIBUTION       │
└────────────────────────────────────────────────────────────┘
```

CONCEPT: Subtraction -- A minus B, 8-bit wraparound
I/O RANGE: $00 -- $02
CYCLE LIMIT: 500
STATUS: LIVE

I/O SPECIFICATION

INPUT	$00	RAW RANGE
INPUT	$01	WIND DRIFT
OUTPUT	$02	CORRECTED RANGE (($00 - $01) mod 256)

Compute ($00 - $01) mod 256, store the low byte in $02. Do not modify $00 or $01. Terminate with BRK.

NEW INSTRUCTIONS

```
LDA    -- Load Accumulator
STA    -- Store Accumulator
SEC    -- Set Carry (required before SBC)
SBC    -- Subtract with Carry
BRK    -- Halt execution
```

WORKING KNOWLEDGE

6502 subtraction is A - M - (1-C). SEC sets C=1 so the borrow term is zero: effectively A - M. Always SEC before SBC in single-subtraction programs.

If $00 < $01, the result wraps mod 256 -- this is correct behaviour, not an error condition. The 8-bit result is what the harness expects. No clamping is required.

BRK must be reached for tests to run. A program that loops forever or executes BRK on a wrong path will fail.

```
┌─ 1B — SIGNAL BOOST -- ADDITIVE CORRECTION ── CLUSTER 0 ──┐
│  CLASSIFICATION: TRAINING // AUTHORISED DISTRIBUTION      │
└──────────────────────────────────────────────────────────┘
```

CONCEPT: Addition -- A plus offset
I/O RANGE: $00 -- $02
CYCLE LIMIT: 500
STATUS: LIVE (private build)

I/O SPECIFICATION

INPUT	$00	SIGNAL LEVEL
INPUT	$01	CALIBRATION OFFSET
OUTPUT	$02	BOOSTED SIGNAL

Compute ($00 + $01) mod 256, store in $02. Do not modify $00 or $01. Terminate with BRK.

NEW INSTRUCTIONS

CLC -- Clear Carry (required before ADC)
ADC -- Add with Carry

WORKING KNOWLEDGE

ADC adds A + M + C. CLC clears the carry so the addition is A + M without a stray carry from a previous operation.

The result is 8-bit modular (wraps at $FF). The spec says mod 256 -- do not clamp.

```
┌─ P2 — IMPACT CLASSIFIER -- PROTO B ──────── CLUSTER 0 ──────┐
│  CLASSIFICATION: TRAINING // AUTHORISED DISTRIBUTION        │
└─────────────────────────────────────────────────────────────┘
```

CONCEPT: Conditional branching -- CMP + BCS/BCC
I/O RANGE: $00 -- $03
CYCLE LIMIT: 500
STATUS: LIVE

I/O SPECIFICATION

```
INPUT   $00   RAW RANGE
INPUT   $01   WIND DRIFT
OUTPUT  $03   CLASSIFICATION BYTE
              #$FF = STABLE   ($00 >= $01)
              #$00 = UNSTABLE ($00 <  $01)
```

Compare $00 against $01. Store #$FF in $03 if $00 >= $01, otherwise store #$00 in $03. Do not modify $00 or $01. Terminate with BRK.

NEW INSTRUCTIONS

```
CMP    -- Compare Accumulator (does not modify A)
BCS    -- Branch if Carry Set (A >= M, unsigned)
BCC    -- Branch if Carry Clear (A < M, unsigned)
```

WORKING KNOWLEDGE

CMP sets flags based on A - M but does not store the result. After CMP: C=1 means A >= M (unsigned). C=0 means A < M.

BCS and BCC branch to a label when the condition is met. They do not fall through if the condition is false -- the next instruction executes instead.

```
┌─ P3 — ANOMALY SUPPRESSION FILTER -- PROTO C — CLUSTER 0 ──┐
│  CLASSIFICATION: TRAINING // AUTHORISED DISTRIBUTION      │
└───────────────────────────────────────────────────────────┘
```

CONCEPT: Multiple outputs, borrow detection, clamping
I/O RANGE: $00 -- $03
CYCLE LIMIT: 500
STATUS: LIVE

I/O SPECIFICATION

INPUT	$00	PRIMARY VALUE
INPUT	$01	OFFSET VALUE
OUTPUT	$02	CORRECTED VALUE (0 if suppressed)
OUTPUT	$03	SUPPRESS FLAG (#$01 if suppressed, #$00 if not)

Compute ($00 - $01). If the result is negative (a borrow occurred), store #$00 in $02 and #$01 in $03. Otherwise store the result in $02 and #$00 in $03. Do not modify $00 or $01. Terminate with BRK.

NEW INSTRUCTIONS

BMI -- Branch if Minus (N=1, result was negative)

WORKING KNOWLEDGE

Two outputs require two STA instructions at different addresses. The branch condition determines which pair of values gets stored.

BMI fires when bit 7 of the most recent result is 1 (negative in two's complement). After SBC with a wrap, N will be set.

```
┌─ 3B — DATA RELAY -- REGISTER STAGING ──── CLUSTER 1 ──────┐
│  CLASSIFICATION: PROCEDURAL // FIELD ISSUE                │
└───────────────────────────────────────────────────────────┘
```

CONCEPT: Indexed addressing -- introducing X register
I/O RANGE: $00 -- $05
CYCLE LIMIT: 500
STATUS: LIVE (private build)

I/O SPECIFICATION

INPUT	$00	DATA[0]
INPUT	$01	DATA[1]
INPUT	$02	DATA[2]
OUTPUT	$03	OUT[0]
OUTPUT	$04	OUT[1]
OUTPUT	$05	OUT[2]

Copy each input to its corresponding output using the X register as an index. Direct copy without indexing is not authorised. Terminate with BRK.

NEW INSTRUCTIONS

```
LDX    -- Load X Register
INX    -- Increment X
LDA $nn,X   -- Zero-page indexed by X
STA $nn,X   -- Zero-page indexed by X
```

WORKING KNOWLEDGE

LDA $00,X reads from address ($00 + X). With X=0, reads $00; X=1 reads $01; and so on.

STA $03,X writes to address ($03 + X).

A loop with INX and a branch can process multiple bytes with one code path.

```
┌─ P4 — RANGE TABLE CORRECTION UNIT ──────── CLUSTER 1 ─────┐
│  CLASSIFICATION: PROCEDURAL // FIELD ISSUE                 │
└────────────────────────────────────────────────────────────┘
```

CONCEPT: X/Y registers, indexed loops
I/O RANGE: $00 -- $07
CYCLE LIMIT: 300
STATUS: LIVE

I/O SPECIFICATION

INPUT $00 -- $03 INPUT TABLE (4 raw field readings)
OUTPUT $04 -- $07 CORRECTED TABLE

Add the calibration constant #$0A to each entry at $00--$03 and store the result at the corresponding address in $04--$07. The result wraps mod 256. Use a counted loop. Do not unroll. Terminate with BRK.

NEW INSTRUCTIONS

LDY -- Load Y Register
INY -- Increment Y
LDA $nn,X -- zero-page indexed (review)
CPX -- Compare X Register

WORKING KNOWLEDGE

The correction loop iterates X (or Y) from 0 to 3. Each pass: load from $00,X, ADC #$0A (with CLC before the loop), store to $04,X, advance the index, branch back if not done.

Cycle limit is 300 -- tight. Avoid redundant loads of the constant. Initialise CLC once before the loop, not inside.

```
┌─ P5 — THRESHOLD DENSITY SCANNER ──────────── CLUSTER 1 ──────┐
│  CLASSIFICATION: PROCEDURAL // FIELD ISSUE                   │
└──────────────────────────────────────────────────────────────┘
```

CONCEPT: Indexed scan, count pattern
I/O RANGE: $00 -- $09
CYCLE LIMIT: 500
STATUS: LIVE

I/O SPECIFICATION

INPUT	$00 -- $07	DATA BUFFER (8 bytes)
INPUT	$08	BUFFER LENGTH (fixed at #$08)
OUTPUT	$09	COUNT OF VALUES > $80

Scan the buffer using indexed addressing. Count how many values strictly exceed $80. Store the count in $09. Terminate with BRK.

NEW INSTRUCTIONS

BNE -- Branch if Not Equal (Z=0)
BEQ -- Branch if Equal (Z=1)

WORKING KNOWLEDGE

A counting loop: initialise a count register (or memory location) to zero. For each byte, CMP against #$80; BCS on values >= $81 and increment the count there. Use INX to advance the loop counter.

CMP sets Z=1 when A equals M. CMP does not tell you ``above'' by itself -- read C: C=1 means A >= M.

Read length from $08 once before the loop. The harness fixes it at $08, but reading it lets the same code adapt if the contract widens later.

```
┌─ P6 — SIGNAL INTEGRITY -- CHECKSUM MODULE — CLUSTER 1 ────┐
│  CLASSIFICATION: PROCEDURAL // FIELD ISSUE                │
└───────────────────────────────────────────────────────────┘
```

CONCEPT: Multi-byte addition, sentinel detection
I/O RANGE: $00 -- $05 (packet), outputs at $06/$07
CYCLE LIMIT: 600
STATUS: LIVE

I/O SPECIFICATION

INPUT	$00 -- $05	SIGNAL PACKET (6 bytes, fixed length)
OUTPUT	$06	CHECKSUM (sum of all 6 bytes mod 256)
OUTPUT	$07	SENTINEL FLAG

Sum all six bytes at $00--$05. Store the low byte of the sum in $06. If any byte equals #$FF -- a reserved sentinel value -- store #$01 in $07; otherwise store #$00. The sentinel flag fires for ANY byte that matches, not as a buffer terminator. Sum every byte regardless. Terminate with BRK.

NEW INSTRUCTIONS

(none new -- review ADC, CMP, BNE, LDA $nn,X)

WORKING KNOWLEDGE

ADC adds to A with carry. For a running sum: CLC once before the loop, ADC each byte, let the low 8 bits carry forward. The high byte is discarded (mod 256 result).

Sentinel test runs alongside summation, not as an exit condition. CMP #$FF after each load; if Z=1, set a flag in scratch space. Continue the sum loop either way.

```
┌─ 6B — BITFIELD STATUS -- MASK AND TEST ─── CLUSTER 1 ──────┐
│  CLASSIFICATION: PROCEDURAL // FIELD ISSUE                 │
└────────────────────────────────────────────────────────────┘
```

CONCEPT: AND masking, LSR shifting to extract bits
I/O RANGE: $00 -- $04
CYCLE LIMIT: 400
STATUS: LIVE (private build)

I/O SPECIFICATION

INPUT	$00	STATUS BYTE
OUTPUT	$01	BIT 0 (0 or 1)
OUTPUT	$02	BIT 1 (0 or 1)
OUTPUT	$03	BIT 2 (0 or 1)
OUTPUT	$04	BIT 3 (0 or 1)

Extract bits 0--3 from the status byte at $00. Store each as an isolated 0 or 1 value (not the raw shifted byte). Terminate with BRK.

NEW INSTRUCTIONS

AND -- Bitwise AND
LSR -- Logical Shift Right

WORKING KNOWLEDGE

To extract bit N: shift the byte right N times with LSR, then AND #$01 to isolate the low bit.

Bit 0: AND #$01 directly (no shift needed). Bit 1: LSR A once, then AND #$01. Bit 2: LSR A twice, then AND #$01. Bit 3: LSR A three times, then AND #$01.

Reload from $00 before extracting each bit -- do not try to reuse a partially shifted value.

```
┌─ P7 — PRIORITY ENCODER -- CHANNEL EXTRACTION — CLUSTER 1 ─┐
│  CLASSIFICATION: PROCEDURAL // FIELD ISSUE                │
└───────────────────────────────────────────────────────────┘
```

CONCEPT: AND/LSR nibble extraction, XOR parity
I/O RANGE: $00 (input), $08 -- $0A (outputs)
CYCLE LIMIT: 400
STATUS: LIVE

I/O SPECIFICATION

INPUT	$00	PACKED BYTE
OUTPUT	$08	LOW NIBBLE (bits 0--3)
OUTPUT	$09	HIGH NIBBLE (bits 4--7, shifted to low)
OUTPUT	$0A	PARITY (XOR of bits 0--3 of $00)

Extract low nibble with AND #$0F, store in $08. Extract high nibble by shifting right 4 times then AND #$0F, store in $09. Store parity of the low nibble in $0A (XOR of all four low bits -- result is $00 or $01). Do not modify $00. Terminate with BRK.

NEW INSTRUCTIONS

EOR -- Bitwise Exclusive OR (XOR)

WORKING KNOWLEDGE

Low nibble extraction: AND #$0F masks bits 4--7 to zero, leaving bits 0--3 intact.

High nibble extraction: LSR four times (bit 4 moves to bit 0) then AND #$0F to zero any residual bits.

Parity of a nibble: XOR the nibble byte with itself shifted right 1, then right 2, then right 3; AND #$01 at the end. Alternatively: EOR the low and high nibbles together -- if their XOR has even bit-count the parity is $00, otherwise $01. Only the final bit matters.

```
┌─ 7B — DUAL CORRECTION -- SUBROUTINE DRILL — CLUSTER 2 ────┐
│  CLASSIFICATION: PROCEDURAL // FIELD ISSUE                │
└───────────────────────────────────────────────────────────┘
```

CONCEPT: JSR/RTS -- simplest subroutine use
I/O RANGE: $00 -- $04
CYCLE LIMIT: 600
STATUS: LIVE (private build)

I/O SPECIFICATION

INPUT	$00	VALUE A
INPUT	$01	VALUE B
INPUT	$02	CORRECTION OFFSET
OUTPUT	$03	RESULT A (VALUE A + OFFSET)
OUTPUT	$04	RESULT B (VALUE B + OFFSET)

Apply the correction offset to each value using a subroutine. The subroutine must use JSR and RTS. Inline duplication is not authorised. Terminate with BRK.

NEW INSTRUCTIONS

```
JSR    -- Jump to Subroutine (pushes return address)
RTS    -- Return from Subroutine (pops return address)
```

WORKING KNOWLEDGE

JSR pushes the return address (PC - 1) onto the stack and jumps to the label. RTS pops the address and resumes.

The subroutine receives its argument in A (by convention). Return value also in A. The calling code loads A before JSR, then reads A (or stores it) after JSR returns.

Every JSR must be matched by exactly one RTS on each possible execution path.

```
┌─ P8 — CLAMP SUBROUTINE -- REFACTOR ──────── CLUSTER 2 ──────┐
│  CLASSIFICATION: PROCEDURAL // FIELD ISSUE                  │
└─────────────────────────────────────────────────────────────┘
```

CONCEPT: JSR/RTS -- reusable subroutines with logic
I/O RANGE: $00 -- $05 (inputs), $10 -- $12 (outputs)
CYCLE LIMIT: 700
STATUS: LIVE

I/O SPECIFICATION

INPUT	$00 / $01	PAIR 1 (minuend / subtrahend)
INPUT	$02 / $03	PAIR 2 (minuend / subtrahend)
INPUT	$04 / $05	PAIR 3 (minuend / subtrahend)
OUTPUT	$10	($00 - $01) clamped to 0 if negative
OUTPUT	$11	($02 - $03) clamped to 0 if negative
OUTPUT	$12	($04 - $05) clamped to 0 if negative

Refactor the clamp logic from earlier missions into a single subroutine reachable via JSR/RTS. Call it three times against the three input pairs. Inline duplication is rejected -- the review surface scores subroutine structure. Terminate with BRK.

NEW INSTRUCTIONS

PHA -- Push Accumulator (preserve across subroutine)
PLA -- Pull Accumulator

WORKING KNOWLEDGE

A subroutine that takes a parameter in A and returns a result in A is the standard 6502 register-passing convention. The caller: LDA parameter / JSR LABEL / STA result.

If the subroutine needs to use X or Y internally, the subroutine (not the caller) is responsible for preserving and restoring them via the stack. PHA/PLA preserves A. For X/Y use TXA/PHA and PLA/TAX patterns.

```
┌─ P9 — SEQUENCE REVERSAL -- STACK METHOD ── CLUSTER 2 ──────┐
│  CLASSIFICATION: PROCEDURAL // FIELD ISSUE                 │
└────────────────────────────────────────────────────────────┘
```

CONCEPT: PHA/PLA -- stack as data structure
I/O RANGE: $00 -- $14
CYCLE LIMIT: 400
STATUS: LIVE

I/O SPECIFICATION

INPUT	$00--$04	INPUT SEQUENCE (5 bytes)
OUTPUT	$10--$14	REVERSED SEQUENCE

Push all 5 input bytes onto the stack (first to last), then pop them into the output addresses (first to last). The LIFO nature of the stack produces the reversal. Do not modify $00--$04. Terminate with BRK.

NEW INSTRUCTIONS

(none new -- review PHA, PLA, indexed load/store)

WORKING KNOWLEDGE

Push order: $00, $01, $02, $03, $04 (five PHA instructions, loading each byte first).

Pop order: $10, $11, $12, $13, $14 (five PLA/STA pairs). The last byte pushed ($04) is the first popped -- it goes to $10. The first byte pushed ($00) lands at $14.

Stack is LIFO: last in, first out.

```
┌─ 9B — INDIRECT LOAD -- POINTER BASICS ──── CLUSTER 2 ──────┐
│  CLASSIFICATION: RESTRICTED // FIELD ISSUE                 │
└────────────────────────────────────────────────────────────┘
```

CONCEPT: (Indirect),Y with Y=0 -- simplest indirect
I/O RANGE: $0E -- $10
CYCLE LIMIT: 400
STATUS: LIVE (private build)

I/O SPECIFICATION

INPUT	$0E	POINTER LOW BYTE
INPUT	$0F	POINTER HIGH BYTE ($00 for zero-page target)
OUTPUT	$10	VALUE AT POINTED ADDRESS

Load the value from the address stored at $0E/$0F and write it to $10. Use LDA ($0E),Y with Y=0. Terminate with BRK.

NEW INSTRUCTIONS

LDY -- Load Y Register (review: set Y=0)
LDA ($nn),Y -- Indirect indexed by Y

WORKING KNOWLEDGE

LDA ($0E),Y reads a 16-bit address from $0E (low byte) and $0F (high byte), then adds Y to that address, and loads the byte at the resulting address.

With Y=0, the effective address is exactly the value in $0E/$0F. The pointer IS the address -- you are dereferencing a pointer.

$0F will be $00 in this puzzle (target is in the zero page). The effective address is 0x00nn where nn = $0E.

```
┌─ P10 — DUAL POINTER ACCESS -- INDIRECT MODES — CLUSTER 2 ─┐
│  CLASSIFICATION: RESTRICTED // FIELD ISSUE                │
└───────────────────────────────────────────────────────────┘
```

CONCEPT: (Indirect,X) and (Indirect),Y
I/O RANGE: $10--$17 (outputs), $E0--$E3 (pointer table)
CYCLE LIMIT: 500
STATUS: LIVE

I/O SPECIFICATION

INPUT	$E0/$E1	POINTER TO BLOCK A
INPUT	$E2/$E3	POINTER TO BLOCK B
OUTPUT	$10--$13	BLOCK A COPY (4 bytes)
OUTPUT	$14--$17	BLOCK B COPY (4 bytes)

Copy 4 bytes from each source block to the output region. Use LDA ($E0,X) with X=0 to verify indexed-indirect mode. Use LDA ($E0),Y with Y=0--3 to copy block A. Use LDA ($E2),Y with Y=0--3 to copy block B. Terminate with BRK.

NEW INSTRUCTIONS

LDA ($nn,X) -- Indexed indirect
LDA ($nn),Y -- Indirect indexed (review from 9B)

WORKING KNOWLEDGE

LDA ($E0,X) with X=0: adds X to $E0, reads the 16-bit pointer at $E0, uses that address as the effective address.

LDA ($E0),Y with Y=0..3: reads the 16-bit pointer at $E0/$E1, adds Y to get the effective address. Iterating Y walks through consecutive bytes of the block.

The two pointer addresses ($E0 and $E2) are separate. Use the correct one for each block.

```
┌─ P11 — FILTERED COPY WITH MASK ──────────── CLUSTER 2 ──────┐
│  CLASSIFICATION: RESTRICTED // FIELD ISSUE                  │
└─────────────────────────────────────────────────────────────┘
```

CONCEPT: Indirect indexed + AND masking + counters
I/O RANGE: $10--$18 (outputs), $20 (mask), $FE/$FF (pointer)
CYCLE LIMIT: 800
STATUS: LIVE

I/O SPECIFICATION

INPUT	$FE/$FF	POINTER TO SOURCE BLOCK (8 bytes)
INPUT	$20	MASK BYTE
OUTPUT	$10--$17	FILTERED BYTES
OUTPUT	$18	SUPPRESSED COUNT

Read 8 bytes via ($FE),Y for Y=0--7. AND each byte with the mask at $20. Store results in $10--$17. Count bytes that were zeroed by the mask (AND result == $00) and store that count in $18. Terminate with BRK.

NEW INSTRUCTIONS

INC -- Increment Memory (for counters)
DEC -- Decrement Memory

WORKING KNOWLEDGE

The AND mask operation zeros out bits that are 0 in the mask. A byte that becomes $00 after masking means all its set bits were in the zeroed positions of the mask.

To count zeroed results: after each AND, test with BEQ (Z=1 when result is zero), then INC the counter address.

Using a memory byte as a counter: initialise to $00 with LDA #$00 / STA $18, then INC $18 on each match.

```
┌─ 11B — SHIFT REGISTER -- FIELD EXTRACTION — CLUSTER 3 ────┐
│  CLASSIFICATION: RESTRICTED // FIELD ISSUE                │
└───────────────────────────────────────────────────────────┘
```

CONCEPT: LSR + AND -- simple bit field extraction
I/O RANGE: $00 -- $02
CYCLE LIMIT: 400
STATUS: LIVE (private build)

I/O SPECIFICATION

INPUT	$00	PACKED BYTE
OUTPUT	$01	UPPER NIBBLE (shifted to bits 0--3)
OUTPUT	$02	LOWER NIBBLE (masked to bits 0--3)

Shift right 4 times to extract the upper nibble, store in $01. Mask with AND #$0F to extract the lower nibble, store in $02. Terminate with BRK.

NEW INSTRUCTIONS

(none new -- review LSR, AND)

WORKING KNOWLEDGE

Upper nibble: Load $00, LSR four times (bits 7--4 shift to bits 3--0), AND #$0F (mask the result to be safe). Store.

Lower nibble: Load $00 fresh, AND #$0F (zeroes bits 7--4, leaves bits 3--0). Store.

The two extractions are independent. Reload $00 before the lower nibble operation -- do not try to undo shifts.

```
┌─ P12 — SERIAL BIT PACKER ──────────────── CLUSTER 3 ──────┐
│  CLASSIFICATION: RESTRICTED // FIELD ISSUE                │
└───────────────────────────────────────────────────────────┘
```

CONCEPT: ROL/ROR -- bit rotation through carry
I/O RANGE: $00 -- $0F
CYCLE LIMIT: 600
STATUS: LIVE

I/O SPECIFICATION

INPUT	$00--$07	8 INDIVIDUAL BIT VALUES (0 or 1 each)
OUTPUT	$10	PACKED BYTE

Pack 8 single-bit values into one byte. Bit 0 of $00 becomes bit 7 of the result (most significant), working down. Use ROL or ROR to shift bits through the carry into the accumulator. Terminate with BRK.

NEW INSTRUCTIONS

ROL -- Rotate Left through Carry
ROR -- Rotate Right through Carry

WORKING KNOWLEDGE

ROL inserts the old carry into bit 0 and shifts bit 7 out to carry. To pack a bit: load it, check if it is nonzero (CMP #$01 / BCS), set or clear carry accordingly, then ROL the accumulator.

Alternatively: load each input bit into A, transfer its low bit to carry (via CLC/SEC based on bit 0), then rotate the output accumulator. Eight such operations build the full byte.

The order of packing matters -- the first bit loaded ends up in the highest position after 8 rotations.

```
┌─ P13 — THRESHOLD MATRIX SCAN ──────────── CLUSTER 3 ─────┐
│  CLASSIFICATION: RESTRICTED // FIELD ISSUE                │
└──────────────────────────────────────────────────────────┘
```

CONCEPT: CPX/CPY -- 2D iteration, bit flag assembly
I/O RANGE: $00 -- $0F (4x4 matrix), thresholds at $20/$21, output at $30
CYCLE LIMIT: 1000
STATUS: LIVE

I/O SPECIFICATION

INPUT	$00 -- $0F	MATRIX DATA (4 rows x 4 cols, row-major)
INPUT	$20	ROW THRESHOLD
INPUT	$21	COL THRESHOLD
OUTPUT	$30	BITFIELD RESULT

Scan the matrix. For each cell where row index > $20 AND column index > $21, set the corresponding bit in $30. Bit assignment is fixed by position:

BIT 0 -- cell [2][2] BIT 2 -- cell [3][2]
BIT 1 -- cell [2][3] BIT 3 -- cell [3][3]

Cells outside row 2-3, column 2-3 contribute no bits. Terminate with BRK.

NEW INSTRUCTIONS

CPX -- Compare X Register
CPY -- Compare Y Register

WORKING KNOWLEDGE

2D iteration: outer loop over rows (Y), inner loop over columns (X). Address of cell (row, col) = Y * 4 + X for this 4-wide matrix. Only cells in rows 2--3, cols 2--3 contribute bits to the output; iterating the full grid and gating with CPX/CPY is one valid approach.

CPX and CPY behave identically to CMP but compare X or Y against the operand without modifying the register.

Bit assembly: use ASL or ROL on an accumulator that accumulates

the output byte. Each inner-loop pass contributes one bit.

```
┌─ P14 — IN-PLACE NORMALISATION PASS ──────── CLUSTER 3 ────┐
│  CLASSIFICATION: TERMINAL // ABOVE CONTRACT LEVEL         │
└───────────────────────────────────────────────────────────┘
```

CONCEPT: INC/DEC -- in-place memory modification
I/O RANGE: $00 -- $07 (buffer), $10 (flag byte)
CYCLE LIMIT: 800
STATUS: LIVE

I/O SPECIFICATION

INPUT/OUTPUT	$00 -- $07	DATA BUFFER (modified in place)
OUTPUT	$10	FLAG BYTE

For each byte at $00 -- $07:

IF value > $80	DEC the byte; set bit N in $10 (bit 0 = address $00, bit 7 = $07)
IF value == $00	INC the byte
OTHERWISE	leave unchanged

Apply INC and DEC directly to memory addresses; do not load into A and write back. Terminate with BRK.

NEW INSTRUCTIONS

```
INC $nn    -- Increment memory address directly
DEC $nn    -- Decrement memory address directly
```

WORKING KNOWLEDGE

INC and DEC modify memory without loading into A. The syntax INC $03 increments the byte at address $03 and sets N and Z flags based on the result.

For indexed in-place modification: INC $00,X increments the byte at address $00 + X. Iterate X through the range.

INC/DEC do not affect the carry flag.

```
┌─ P15 — DEAD RECKONING -- POSITION INTEGRATOR — CLUSTER 3 ─┐
│  CLASSIFICATION: TERMINAL // ABOVE CONTRACT LEVEL          │
└────────────────────────────────────────────────────────────┘
```

CONCEPT: Full pipeline: bit decoding, multi-axis math, clamping, signed comparison
I/O RANGE: $00 -- $3F
CYCLE LIMIT: 1200
STATUS: LIVE

I/O SPECIFICATION

INPUT	$00	INITIAL X POSITION
INPUT	$01	INITIAL Y POSITION
INPUT	$10+	8 COMMAND BYTES (direction + magnitude)
OUTPUT	$20	FINAL X
OUTPUT	$21	FINAL Y
OUTPUT	$22	CLAMP COUNT

Each command byte encodes: bits 7--6 = direction (0=N, 1=E, 2=S, 3=W), bits 5--0 = magnitude. Apply each command to the position, clamp the result to 0..$3F on each axis, count clamped operations. Terminate with BRK.

NEW INSTRUCTIONS

(none new -- full set required)

WORKING KNOWLEDGE

Bit extraction for the direction field: shift right 6 times (two LSR) gives bits 7--6 in positions 1--0. Compare with #$00--$03 to determine direction.

Magnitude extraction: AND #$3F isolates bits 5--0.

Signed vs unsigned: positions are unsigned $00--$3F. Underflow (subtracting past $00) is caught by BCC after SBC. Overflow (adding past $3F) is caught by CMP #$3F / BCC or BCS.

This is the final assignment. The full instruction set is required.

```
┌─ H1 — NULL RETURN PROTOCOL ──────────── UNCLASSIFIED ──────┐
│  CLASSIFICATION: ABOVE CONTRACT LEVEL // NEED TO KNOW ONLY  │
└─────────────────────────────────────────────────────────────┘
```

CONCEPT: Indirect JSR via zero-page vector, XOR checksum
I/O RANGE: $00 -- $14
CYCLE LIMIT: 500
STATUS: UNLOCKED (condition undisclosed)

I/O SPECIFICATION

INPUT	$00--$03	FOUR INPUT VALUES
INPUT	$E8/$E9	SUBROUTINE VECTOR (little-endian pointer)
INPUT	$EC	OUTPUT BASE ADDRESS (prime before first call)
OUTPUT	$10--$13	TRANSFORMED VALUES
OUTPUT	$14	CHECKSUM (XOR of $10--$13)

Call the subroutine at the address stored in $E8/$E9 four times, passing each input from $00--$03 in A. Before the first call, store #$10 into $EC. After all four calls, XOR $10--$13 together and store the result at $14. Terminate with BRK.

NEW INSTRUCTIONS

(none new -- full set required)

WORKING KNOWLEDGE

A zero-page vector holds a 16-bit address, little-endian. To dereference it: set Y=0 and use LDA ($E8),Y to read the byte at the pointed-to address.

Indirect subroutine calls: the game's assembler supports indirect JSR using the same syntax as indirect LDA. Use JSR ($E8,X) with X=0 to call the subroutine whose address is stored in $E8/$E9.

XOR checksum: initialise to $00, EOR each byte in sequence. Each EOR leaves the running accumulation.

```
┌─ H2 — VECTOR OVERWRITE -- DEEP ACCESS — UNCLASSIFIED ──────┐
│  CLASSIFICATION: ABOVE CONTRACT LEVEL // NEED TO KNOW ONLY  │
└─────────────────────────────────────────────────────────────┘
```

CONCEPT: Self-modifying pointer, transform pipeline, ROL
I/O RANGE: $20 -- $2B (outputs), $F0 -- $F3 (pointers)
CYCLE LIMIT: 700
STATUS: UNLOCKED (condition undisclosed)

I/O SPECIFICATION

INPUT	$F0/$F1	POINTER TO BLOCK A (6 bytes)
INPUT	$F2/$F3	POINTER TO BLOCK B (6 bytes)
OUTPUT	$20--$25	TRANSFORMED BLOCK A
OUTPUT	$26--$2B	TRANSFORMED BLOCK B

Read 6 bytes from block A via ($F0),Y for Y=0--5. Apply the transformation (XOR #$A5, then ROL once) to each byte. Store results at $20--$25. Self-modify $F2/$F3 as required, then read block B via ($F2),Y for Y=0--5, apply the same transformation, store at $26--$2B. Terminate with BRK.

NEW INSTRUCTIONS

(none new -- full set required)

WORKING KNOWLEDGE

ROL rotates left through carry. The carry flag feeds into bit 0 and bit 7 exits to carry. To ensure a predictable result, set carry to a known state (CLC) before each ROL.

Self-modification: store a new low byte (and if needed high byte) into the pointer pair at $F2/$F3 during execution. Subsequent indirect loads from ($F2),Y will use the updated address.

Transform order: XOR first, then ROL. Applying them in reverse produces a different result.

```
┌─ H3 — FINAL INTEGRATION -- TERMINATION PASS — UNCLASSIFIED ┐
│  CLASSIFICATION: ABOVE CONTRACT LEVEL // NEED TO KNOW ONLY  │
└─────────────────────────────────────────────────────────────┘
```

CONCEPT: 16-bit accumulation, bitfield, self-modifying indirect walk, conditional branching
I/O RANGE: $80 -- $84 (outputs), $C0 -- $CF (records), $F8/$F9 (pointer)
CYCLE LIMIT: 900
STATUS: UNLOCKED (condition undisclosed)

I/O SPECIFICATION

INPUT	$C0--$CF	16-BYTE RECORD BLOCK (8 records, 2 bytes each: Category byte then Count byte, row-major)
INPUT	$F8/$F9	POINTER TO RECORD BLOCK (prime to $C0/$00)
OUTPUT	$80	SUM OF COUNT BYTES (LOW BYTE, flagged records)
OUTPUT	$81	SUM OF COUNT BYTES (HIGH BYTE)
OUTPUT	$82	BITFIELD (bit N = record N is flagged)
OUTPUT	$83	TOTAL RECORDS PROCESSED (always 8)
OUTPUT	$84	COUNT OF FLAGGED RECORDS

For each of 8 records: if the Category byte is >= #$04, add the Count byte to the 16-bit accumulator at $80/$81 and set the corresponding bit in $82. Store 8 in $83 and the flagged count in $84. Use ($F8),Y indirect access; self-modify $F8 after the first 4 records to advance the walk pointer. Terminate with BRK.

NEW INSTRUCTIONS

(none new -- full set required)

WORKING KNOWLEDGE

16-bit addition: add the low byte to $80 with ADC; if carry is set, INC $81. Always CLC before the first ADC in a sequence.

Bitfield assembly: maintain a scratch byte. For each qualifying record N, set bit N with LDA scratch / ORA bitmask / STA scratch. The bitmask for bit N is (1 << N), which can be tracked by starting at $01 and ASL once per record.

Self-modifying pointer: after Y reaches 7 (end of first 4 records with 2 bytes each = offset 8 from base), store the updated address low byte into $F8. The high byte remains $00 if the target is in the zero page.

CLASSIFIED CONTENT

```
┌─ SECTION 5 — CLASSIFIED CONTENT ──────────────────────────┐
│  ANOMALY LAYER // ACCESS RESTRICTED                       │
│  THIS SECTION NOT PART OF STANDARD DISTRIBUTION           │
└───────────────────────────────────────────────────────────┘
```

DOCUMENT STATUS: ABOVE CONTRACT LEVEL
CLASSIFICATION: ZERO-RETENTION
DISTRIBUTION: INTERNAL ONLY

The standard mission sequence comprises 21 assignments.

There are three more.

They are not listed in the mission select screen. They do not appear in the standard assignment queue. Access to each requires a specific action taken during the course of a standard mission -- an action that deviates from the documented I/O contract.

The machine does not prevent deviations. It observes them.

WHAT THE SYSTEM LOGS CONTAIN

At certain mission indices, a SYSTEM LOG becomes accessible after mission completion. These logs are not debrief text. They are routing records, audit entries, process logs.

They were not written for you to read.

If you find one, read it carefully.

ANOMALY CLASSIFICATION

The three unlockable modules are designated:

```
┌──────────────────────────────────────────────────────┐
│  H1  NULL RETURN PROTOCOL                            │
│      Cluster 1 anomaly layer                         │
│      Cycle limit: 500                                │
│                                                      │
│  H2  VECTOR OVERWRITE -- DEEP ACCESS                 │
│      Cluster 2 anomaly layer                         │
│      Cycle limit: 700                                │
│                                                      │
│  H3  FINAL INTEGRATION -- TERMINATION PASS           │
│      Cluster 3 anomaly layer                         │
│      Cycle limit: 900                                │
└──────────────────────────────────────────────────────┘
```

Each is unlocked by a specific anomalous output during the corresponding standard mission. The system detects the anomaly and opens the access channel.

UNLOCK MECHANISM

Each hidden module is unlocked through the standard mission interface. No separate action is required beyond the program you submit to the test harness. The anomaly is in the output -- not in any menu or command sequence.

The system log that appears at each unlock point contains a clue to the hidden mission's objectives. Read the log before attempting the module.

ON WORKING KNOWLEDGE

The three hidden modules require advanced use of the full instruction set, including:

- Indirect addressing in both modes

- Self-modifying code via pointer table writes
- Subroutine call via indirect vectors
- Multi-byte accumulation (16-bit result)
- Bitfield construction

These modules introduce no new instructions. Section 2 documents everything you need.

ADVISORY

[WARN] These modules are not training exercises.
[WARN] The I/O contracts are unusual.
[WARN] Read the narrative text at unlock time.
[WARN] Read it again after you complete the module.

CREDITS AND COLOPHON

```
┌─ SECTION 6 — CREDITS AND COLOPHON ──────────────────────┐
│  ZERO PAGE MICROCOMPUTER                                │
│  OPERATOR REFERENCE MANUAL                              │
└─────────────────────────────────────────────────────────┘
```

BUILD INFORMATION

PRODUCT: ZERO PAGE
VERSION: 0.1.0
PLATFORM: Electron (Steam target)
ARCHITECTURE: MOS 6502 subset simulator
ENGINE: Custom JavaScript VM

MANUAL REV: 1.0
MANUAL DATE: 2026

DESIGNED AND OPERATED BY

Ray Weiss

PROJECT LINKS

itch.io: lerugray.itch.io/zero-page
github: github.com/lerugray/ZERO-PAGE

LICENSE

The ZERO PAGE game, manual, and all associated content are copyright Ray Weiss. All rights reserved.

The public web build is available at the github link above. The Electron/Steam build containing the full 21-mission campaign and bridge puzzles is the version documented in this manual.

ACKNOWLEDGEMENTS

This game is a love letter to the Zachtronics library, particularly TIS-100 (2015). The design vocabulary -- phosphor terminal, monospace everything, puzzle-as-spec- sheet, no hand-holding -- is inherited directly from that tradition. Build it right or not at all.

TYPOGRAPHIC NOTE

This manual is set in Consolas in phosphor green on near-black. Box-drawing glyphs (┌ │ ─ ├) carry the section dividers. No serifs. No gradients. One font. The terminal speaks one language.

```
┌──────────────────────────────────────────┐
│  END OF DOCUMENT                         │
│  ZERO PAGE INITIATIVE                    │
│  FIELD OPERATIONS DIVISION               │
│  CLASSIFICATION: FIELD ISSUE             │
│  DOCUMENT DOES NOT REQUIRE RETURN        │
└──────────────────────────────────────────┘
```