THE LOGIC BENCH 6502

# Reference Manual

The instruction set, architecture, and addressing modes of the MOS 6502 as built and run inside Logic Bench.

Logic Bench · every box opens, nothing inside is fake

This manual documents the **6502 you can load, run, and take apart in Logic Bench** — the behavioral model and the gate-built version alike, which share one instruction set. That instruction set is the full documented NMOS 6502: **56 mnemonics across 13 addressing modes — 151 opcodes in all**, the same set that shipped in the Apple II, the Commodore 64, the BBC Micro and the NES.

Everything here is generated from Logic Bench's own ISA table and verified against its gold-oracle emulator (proven against the Klaus Dörmann functional suite — over 30 million instructions, zero errors). So this is not a generic 6502 reference copied from elsewhere: it describes *this* machine, the one on your bench, exactly as it behaves.

How to read it: skim the **Architecture** to meet the registers and memory map, learn the **Addressing Modes** (the 6502's defining idea), then use the **Instruction Reference** as a lookup — every instruction with what it does, which flags it touches, and the opcode byte for each addressing mode it supports.

---

## Architecture

### Registers

The 6502 has a small set of registers — and you have built every one of them in the workbook.

- **A — Accumulator (8-bit).** The working register: arithmetic, logic, loads and stores flow through A. It is the one register the ALU operates on.
- **X, Y — Index registers (8-bit).** Counters and offsets. They drive the indexed addressing modes (`$8000,X`), step through tables and loops, and increment/decrement on their own (INX, DEY).
- **S — Stack pointer (8-bit).** Points to the top of the stack, which lives in **page one** (`$0100`–`$01FF`). It counts *down* on a push and *up* on a pull. On reset it is wound down to `$FD`.
- **PC — Program counter (16-bit).** The address of the next instruction. Because it is 16 bits wide, the 6502 can address the full **64 KB** of memory.
- **P — Processor status (8-bit).** The flags, described below.

### The status register (P)

P holds seven meaningful flag bits. Most instructions update **N** and **Z** as a side effect; arithmetic and shifts also touch **C** and **V**.

| Bit | Flag | Name | Meaning |
|---|---|---|---|
| 7 | **N** | Negative | Bit 7 of the last result (its sign). |
| 6 | **V** | Overflow | Signed overflow from ADC/SBC; bit 6 from BIT. |
| 5 | — | (unused) | Always reads as 1. |
| 4 | **B** | Break | Distinguishes BRK from a hardware IRQ on the stacked status. |
| 3 | **D** | Decimal | When set, ADC/SBC work in binary-coded decimal. |
| 2 | **I** | Interrupt | When set, maskable IRQs are ignored. |
| 1 | **Z** | Zero | Set when the last result was zero. |
| 0 | **C** | Carry | Carry/borrow out of arithmetic; the bit shifted out by shifts/rotates. |

### Memory map

The 6502 sees a flat 64 KB address space. A few regions have fixed meaning:

| Range | Use |
|---|---|
| `$0000`–`$00FF` | **Zero page** — special fast addressing modes live here; precious space. |
| `$0100`–`$01FF` | **Stack** (page one) — JSR/RTS return addresses and PHA/PLA data. |
| `$0200`–`$7FFF` | General RAM — where Logic Bench programs are assembled (the demos `.org $0200`). |
| `$8000`–`$BFFF` | **Screen** — the 32×32 framebuffer on the behavioral/gate boards; each `STA` here lights pixels. |
| `$D000`–`$D015` | **Calculator I/O** (Pocket Calculator board only) — `$D000` keypad column-select, `$D001` keypad rows, `$D010`–`$D015` the six display digits. |
| `$FFFA`–`$FFFF` | **Vectors** — three 16-bit pointers the CPU reads on reset and interrupts. |

A note on memory maps: these regions are *board conventions*, not fixed properties of the 6502 — the CPU sees a flat 64 KB space and a memory-mapped device lives wherever the board's address decoder puts it. The screen at `$8000` belongs to the graphics/ball boards; the `$D0xx` I/O block belongs to the **Pocket Calculator** board, whose gate-built I/O controller decodes that page to a 4×4 keypad and a 6-digit display (drill into it on the bench to see the decoder, the latches, and the keypad-row mux).

### Reset and interrupt vectors

The 6502 does not begin executing at address 0. On power-up it reads a 16-bit address from the **reset vector** and jumps there — exactly as the gate-built board boots:

| Vector | Address | When the CPU reads it |
|---|---|---|
| **NMI** | `$FFFA/$FFFB` | A non-maskable interrupt occurs. |
| **RESET** | `$FFFC/$FFFD` | Power-on / reset — execution begins here. |
| **IRQ/BRK** | `$FFFE/$FFFF` | A maskable IRQ, or a BRK instruction. |

---

## Addressing Modes

An *addressing mode* is how an instruction names the data it works on. The 6502's range of modes is its defining feature — the same `LDA` can take a constant, a memory cell, or a computed table entry, depending on the mode. Each mode below is a different way of answering "where is the operand?"

- **Implied / Accumulator** — no operand, or the accumulator itself (`INX`, `ASL A`).
- **Immediate** (`#$nn`) — the operand *is* the byte in the instruction; a constant.
- **Zero Page** (`$nn`) — an address in `$0000`–`$00FF`; one byte shorter and faster.
- **Zero Page,X / ,Y** (`$nn,X`) — zero-page address plus an index register (wraps within page 0).
- **Absolute** (`$nnnn`) — a full 16-bit address anywhere in memory.
- **Absolute,X / ,Y** (`$nnnn,X`) — a 16-bit base plus an index; the workhorse for walking tables and the framebuffer.

- **(Indirect,X)** ( `($nn,X)` ) — a pointer *table* in zero page, indexed by X, then followed.
- **(Indirect),Y** ( `($nn),Y` ) — a pointer in zero page, followed, then offset by Y; the classic way to walk a 16-bit pointer through data.
- **Indirect** ( `($nnnn)` ) — used only by `JMP`: jump through a pointer. (This machine faithfully reproduces the famous `$xxFF` page-boundary quirk.)
- **Relative** ( `label` ) — branches only; a signed offset (–128..+127) from the next instruction.

| Mode | Syntax | Bytes | Example |
|---|---|---|---|
| Implied | `` | 1 | `INX` |
| Accumulator | `A` | 1 | `ASL A` |
| Immediate | `#$nn` | 2 | `LDA #$2A` |
| Zero Page | `$nn` | 2 | `LDA $40` |
| Zero Page,X | `$nn,X` | 2 | `LDA $40,X` |
| Zero Page,Y | `$nn,Y` | 2 | `LDX $40,Y` |
| (Indirect,X) | `($nn,X)` | 2 | `LDA ($40,X)` |
| (Indirect),Y | `($nn),Y` | 2 | `LDA ($40),Y` |
| Relative | `$nnnn (label)` | 2 | `BNE loop` |
| Absolute | `$nnnn` | 3 | `LDA $8000` |
| Absolute,X | `$nnnn,X` | 3 | `STA $8000,X` |
| Absolute,Y | `$nnnn,Y` | 3 | `LDA $8000,Y` |
| Indirect | `($nnnn)` | 3 | `JMP ($FFFC)` |

---

## Instruction Reference

All 56 mnemonics, grouped by purpose. Each entry lists what the instruction does, the flags it affects, and the opcode byte for every addressing mode it supports.

**Reading the flag line:** the eight positions are **N V – B D I Z C**. A letter or `0` / `1` means the instruction affects that flag (forces it low/high); `–` means it leaves the flag

untouched. For example, `N – – – – – Z –` means "updates N and Z, leaves the rest alone."

## Load / Store

### LDA — Load Accumulator

Loads a byte from the operand into the accumulator A. Sets N from bit 7 of the value and Z if the value is zero.

**Flags:** `N – – – – – Z –`

| Mode | Syntax | Opcode | Bytes |
|---|---|---|---|
| Immediate | `LDA #$nn` | `$A9` | 2 |
| Zero Page | `LDA $nn` | `$A5` | 2 |
| Zero Page,X | `LDA $nn,X` | `$B5` | 2 |
| (Indirect,X) | `LDA ($nn,X)` | `$A1` | 2 |
| (Indirect),Y | `LDA ($nn),Y` | `$B1` | 2 |
| Absolute | `LDA $nnnn` | `$AD` | 3 |
| Absolute,X | `LDA $nnnn,X` | `$BD` | 3 |
| Absolute,Y | `LDA $nnnn,Y` | `$B9` | 3 |

### LDX — Load X Register

Loads a byte from the operand into index register X. Sets N and Z from the loaded value.

**Flags:** `N – – – – – Z –`

| Mode | Syntax | Opcode | Bytes |
|---|---|---|---|
| Immediate | `LDX #$nn` | `$A2` | 2 |
| Zero Page | `LDX $nn` | `$A6` | 2 |
| Zero Page,Y | `LDX $nn,Y` | `$B6` | 2 |
| Absolute | `LDX $nnnn` | `$AE` | 3 |
| Absolute,Y | `LDX $nnnn,Y` | `$BE` | 3 |

### LDY — Load Y Register

Loads a byte from the operand into index register Y. Sets N and Z from the loaded value.

**Flags:** `N – – – – – Z –`

| Mode | Syntax | Opcode | Bytes |
|---|---|---|---|
| Immediate | `LDY #$nn` | `$A0` | 2 |
| Zero Page | `LDY $nn` | `$A4` | 2 |
| Zero Page,X | `LDY $nn,X` | `$B4` | 2 |
| Absolute | `LDY $nnnn` | `$AC` | 3 |
| Absolute,X | `LDY $nnnn,X` | `$BC` | 3 |

### STA — Store Accumulator

Stores the accumulator A into the addressed memory location. Affects no flags. This is how a program writes to RAM, to memory-mapped I/O, and to the framebuffer.

**Flags:** `– – – – – – – –`

| Mode | Syntax | Opcode | Bytes |
|---|---|---|---|
| Zero Page | `STA $nn` | `$85` | 2 |
| Zero Page,X | `STA $nn,X` | `$95` | 2 |
| (Indirect,X) | `STA ($nn,X)` | `$81` | 2 |
| (Indirect),Y | `STA ($nn),Y` | `$91` | 2 |
| Absolute | `STA $nnnn` | `$8D` | 3 |
| Absolute,X | `STA $nnnn,X` | `$9D` | 3 |
| Absolute,Y | `STA $nnnn,Y` | `$99` | 3 |

### STX — Store X Register

Stores index register X into the addressed memory location. Affects no flags.

**Flags:** `– – – – – – – –`

| Mode | Syntax | Opcode | Bytes |
|---|---|---|---|
| Zero Page | `STX $nn` | `$86` | 2 |
| Zero Page,Y | `STX $nn,Y` | `$96` | 2 |
| Absolute | `STX $nnnn` | `$8E` | 3 |

### STY — Store Y Register

Stores index register Y into the addressed memory location. Affects no flags.

**Flags:** `– – – – – – – –`

| Mode | Syntax | Opcode | Bytes |
|---|---|---|---|
| Zero Page | `STY $nn` | `$84` | 2 |
| Zero Page,X | `STY $nn,X` | `$94` | 2 |
| Absolute | `STY $nnnn` | `$8C` | 3 |

# Register Transfers

## TAX — Transfer A to X

Copies the accumulator into X. Sets N and Z from the value moved.

**Flags:** `N – – – – – Z –`

| Mode | Syntax | Opcode | Bytes |
|---|---|---|---|
| Implied | `TAX` | `$AA` | 1 |

## TAY — Transfer A to Y

Copies the accumulator into Y. Sets N and Z from the value moved.

**Flags:** `N – – – – – Z –`

| Mode | Syntax | Opcode | Bytes |
|---|---|---|---|
| Implied | `TAY` | `$A8` | 1 |

## TXA — Transfer X to A

Copies X into the accumulator. Sets N and Z from the value moved.

**Flags:** `N – – – – – Z –`

| Mode | Syntax | Opcode | Bytes |
|---|---|---|---|
| Implied | `TXA` | `$8A` | 1 |

## TYA — Transfer Y to A

Copies Y into the accumulator. Sets N and Z from the value moved.

**Flags:** `N – – – – – Z –`

| Mode | Syntax | Opcode | Bytes |
|---|---|---|---|
| Implied | `TYA` | `$98` | 1 |

### TSX — Transfer Stack Pointer to X

Copies the stack pointer S into X. Sets N and Z from the value moved. The only way to read the stack pointer.

**Flags:** `N – – – – – Z –`

| Mode | Syntax | Opcode | Bytes |
|---|---|---|---|
| Implied | `TSX` | `$BA` | 1 |

### TXS — Transfer X to Stack Pointer

Copies X into the stack pointer S. Affects no flags (unlike the other transfers). Used to initialise the stack, typically with LDX #$FF / TXS.

**Flags:** `– – – – – – – –`

| Mode | Syntax | Opcode | Bytes |
|---|---|---|---|
| Implied | `TXS` | `$9A` | 1 |

## Stack

### PHA — Push Accumulator

Pushes A onto the stack (page 1) and decrements S. Affects no flags.

**Flags:** `– – – – – – – –`

| Mode | Syntax | Opcode | Bytes |
|---|---|---|---|
| Implied | `PHA` | `$48` | 1 |

### PHP — Push Processor Status

Pushes the status register P onto the stack with the B bit set, and decrements S. Affects no flags.

**Flags:** `– – – – – – – –`

| Mode | Syntax | Opcode | Bytes |
|---|---|---|---|
| Implied | `PHP` | `$08` | 1 |

### PLA — Pull Accumulator

Increments S and pulls a byte from the stack into A. Sets N and Z from the pulled value.

**Flags:** `N – – – – – Z –`

| Mode | Syntax | Opcode | Bytes |
|---|---|---|---|
| Implied | `PLA` | `$68` | 1 |

### PLP — Pull Processor Status

Increments S and pulls a byte from the stack into the status register P, restoring all flags at once.

**Flags:** `N V x x D I Z C`

| Mode | Syntax | Opcode | Bytes |
|---|---|---|---|
| Implied | `PLP` | `$28` | 1 |

## Arithmetic

### ADC — Add with Carry

Adds the operand and the carry flag to the accumulator: A = A + M + C. Sets C on unsigned overflow (carry out of bit 7) and V on signed overflow. With the D flag set, performs NMOS-accurate binary-coded-decimal addition.

**Flags:** `N V – – – – Z C`

| Mode | Syntax | Opcode | Bytes |
|---|---|---|---|
| Immediate | `ADC #$nn` | `$69` | 2 |
| Zero Page | `ADC $nn` | `$65` | 2 |
| Zero Page,X | `ADC $nn,X` | `$75` | 2 |
| (Indirect,X) | `ADC ($nn,X)` | `$61` | 2 |
| (Indirect),Y | `ADC ($nn),Y` | `$71` | 2 |
| Absolute | `ADC $nnnn` | `$6D` | 3 |
| Absolute,X | `ADC $nnnn,X` | `$7D` | 3 |
| Absolute,Y | `ADC $nnnn,Y` | `$79` | 3 |

### SBC — Subtract with Carry

Subtracts the operand and the borrow (NOT carry) from the accumulator: A = A - M - (1-C). Set C first (SEC) for a plain subtract. Sets V on signed overflow. With D set, performs NMOS-accurate BCD subtraction (flags from the binary result).

**Flags:** `N V – – – – Z C`

| Mode | Syntax | Opcode | Bytes |
|---|---|---|---|
| Immediate | `SBC #$nn` | `$E9` | 2 |
| Zero Page | `SBC $nn` | `$E5` | 2 |
| Zero Page,X | `SBC $nn,X` | `$F5` | 2 |
| (Indirect,X) | `SBC ($nn,X)` | `$E1` | 2 |
| (Indirect),Y | `SBC ($nn),Y` | `$F1` | 2 |
| Absolute | `SBC $nnnn` | `$ED` | 3 |
| Absolute,X | `SBC $nnnn,X` | `$FD` | 3 |
| Absolute,Y | `SBC $nnnn,Y` | `$F9` | 3 |

### INC — Increment Memory

Adds one to the addressed memory byte (read-modify-write). Sets N and Z from the result. Does not affect carry.

**Flags:** `N – – – – – Z –`

| Mode | Syntax | Opcode | Bytes |
|---|---|---|---|
| Zero Page | `INC $nn` | `$E6` | 2 |
| Zero Page,X | `INC $nn,X` | `$F6` | 2 |
| Absolute | `INC $nnnn` | `$EE` | 3 |
| Absolute,X | `INC $nnnn,X` | `$FE` | 3 |

### DEC — Decrement Memory

Subtracts one from the addressed memory byte (read-modify-write). Sets N and Z from the result. Does not affect carry.

**Flags:** `N – – – – – Z –`

| Mode | Syntax | Opcode | Bytes |
|---|---|---|---|
| Zero Page | `DEC $nn` | `$C6` | 2 |
| Zero Page,X | `DEC $nn,X` | `$D6` | 2 |
| Absolute | `DEC $nnnn` | `$CE` | 3 |
| Absolute,X | `DEC $nnnn,X` | `$DE` | 3 |

### INX — Increment X

Adds one to X, wrapping 255 to 0. Sets N and Z from the result.

**Flags:** `N – – – – – Z –`

| Mode | Syntax | Opcode | Bytes |
|---|---|---|---|
| Implied | `INX` | `$E8` | 1 |

### INY — Increment Y

Adds one to Y, wrapping 255 to 0. Sets N and Z from the result.

**Flags:** `N – – – – – Z –`

| Mode | Syntax | Opcode | Bytes |
|---|---|---|---|
| Implied | `INY` | `$C8` | 1 |

### DEX — Decrement X

Subtracts one from X, wrapping 0 to 255. Sets N and Z from the result.

**Flags:** `N – – – – – Z –`

| Mode | Syntax | Opcode | Bytes |
|---|---|---|---|
| Implied | `DEX` | `$CA` | 1 |

### DEY — Decrement Y

Subtracts one from Y, wrapping 0 to 255. Sets N and Z from the result.

**Flags:** `N – – – – – Z –`

| Mode | Syntax | Opcode | Bytes |
|---|---|---|---|
| Implied | `DEY` | `$88` | 1 |

## Logic

### AND — Logical AND

Bitwise ANDs the operand into the accumulator. Sets N and Z from the result.

**Flags:** `N – – – – – Z –`

| Mode | Syntax | Opcode | Bytes |
|---|---|---|---|
| Immediate | `AND #$nn` | `$29` | 2 |
| Zero Page | `AND $nn` | `$25` | 2 |
| Zero Page,X | `AND $nn,X` | `$35` | 2 |
| (Indirect,X) | `AND ($nn,X)` | `$21` | 2 |
| (Indirect),Y | `AND ($nn),Y` | `$31` | 2 |
| Absolute | `AND $nnnn` | `$2D` | 3 |
| Absolute,X | `AND $nnnn,X` | `$3D` | 3 |
| Absolute,Y | `AND $nnnn,Y` | `$39` | 3 |

### ORA — Logical OR

Bitwise ORs the operand into the accumulator. Sets N and Z from the result.

**Flags:** `N – – – – – Z –`

| Mode | Syntax | Opcode | Bytes |
|---|---|---|---|
| Immediate | `ORA #$nn` | `$09` | 2 |
| Zero Page | `ORA $nn` | `$05` | 2 |
| Zero Page,X | `ORA $nn,X` | `$15` | 2 |
| (Indirect,X) | `ORA ($nn,X)` | `$01` | 2 |
| (Indirect),Y | `ORA ($nn),Y` | `$11` | 2 |
| Absolute | `ORA $nnnn` | `$0D` | 3 |
| Absolute,X | `ORA $nnnn,X` | `$1D` | 3 |
| Absolute,Y | `ORA $nnnn,Y` | `$19` | 3 |

### EOR — Exclusive OR

Bitwise exclusive-ORs the operand into the accumulator. Sets N and Z from the result. EOR #$FF inverts every bit of A.

**Flags:** `N – – – – – Z –`

| Mode | Syntax | Opcode | Bytes |
|---|---|---|---|
| Immediate | `EOR #$nn` | `$49` | 2 |
| Zero Page | `EOR $nn` | `$45` | 2 |
| Zero Page,X | `EOR $nn,X` | `$55` | 2 |
| (Indirect,X) | `EOR ($nn,X)` | `$41` | 2 |
| (Indirect),Y | `EOR ($nn),Y` | `$51` | 2 |
| Absolute | `EOR $nnnn` | `$4D` | 3 |
| Absolute,X | `EOR $nnnn,X` | `$5D` | 3 |
| Absolute,Y | `EOR $nnnn,Y` | `$59` | 3 |

### BIT — Bit Test

ANDs A with the operand to set Z (without changing A), and copies bits 7 and 6 of the operand into N and V. Used to test bits and to read status registers non-destructively.

**Flags:** `N V – – – – Z –`

| Mode | Syntax | Opcode | Bytes |
|---|---|---|---|
| Zero Page | `BIT $nn` | `$24` | 2 |
| Absolute | `BIT $nnnn` | `$2C` | 3 |

## Shift / Rotate

### ASL — Arithmetic Shift Left

Shifts the operand (A or memory) left one bit; bit 7 goes into carry, bit 0 becomes 0. Sets N and Z from the result. A fast multiply-by-two.

**Flags:** `N – – – – – Z C`

| Mode | Syntax | Opcode | Bytes |
|---|---|---|---|
| Accumulator | `ASL A` | `$0A` | 1 |
| Zero Page | `ASL $nn` | `$06` | 2 |
| Zero Page,X | `ASL $nn,X` | `$16` | 2 |
| Absolute | `ASL $nnnn` | `$0E` | 3 |
| Absolute,X | `ASL $nnnn,X` | `$1E` | 3 |

### LSR — Logical Shift Right

Shifts the operand right one bit; bit 0 goes into carry, bit 7 becomes 0. N is always cleared. A fast unsigned divide-by-two.

**Flags:** `0 – – – – – Z C`

| Mode | Syntax | Opcode | Bytes |
|---|---|---|---|
| Accumulator | `LSR A` | `$4A` | 1 |
| Zero Page | `LSR $nn` | `$46` | 2 |
| Zero Page,X | `LSR $nn,X` | `$56` | 2 |
| Absolute | `LSR $nnnn` | `$4E` | 3 |
| Absolute,X | `LSR $nnnn,X` | `$5E` | 3 |

### ROL — Rotate Left

Rotates the operand left one bit through carry: the old carry enters bit 0, bit 7 exits to carry. Sets N and Z from the result.

**Flags:** `N – – – – – Z C`

| Mode | Syntax | Opcode | Bytes |
|---|---|---|---|
| Accumulator | `ROL A` | `$2A` | 1 |
| Zero Page | `ROL $nn` | `$26` | 2 |
| Zero Page,X | `ROL $nn,X` | `$36` | 2 |
| Absolute | `ROL $nnnn` | `$2E` | 3 |
| Absolute,X | `ROL $nnnn,X` | `$3E` | 3 |

### ROR — Rotate Right

Rotates the operand right one bit through carry: the old carry enters bit 7, bit 0 exits to carry. Sets N and Z from the result.

**Flags:** `N – – – – – Z C`

| Mode | Syntax | Opcode | Bytes |
|---|---|---|---|
| Accumulator | `ROR A` | `$6A` | 1 |
| Zero Page | `ROR $nn` | `$66` | 2 |
| Zero Page,X | `ROR $nn,X` | `$76` | 2 |
| Absolute | `ROR $nnnn` | `$6E` | 3 |
| Absolute,X | `ROR $nnnn,X` | `$7E` | 3 |

## Compare

### CMP — Compare Accumulator

Computes A - M (without storing) to set the flags: C if A >= M, Z if A == M, N from bit 7 of the difference. Pair with a branch (BEQ/BNE/BCS/BCC) to test.

**Flags:** `N – – – – – Z C`

| Mode | Syntax | Opcode | Bytes |
|---|---|---|---|
| Immediate | `CMP #$nn` | `$C9` | 2 |
| Zero Page | `CMP $nn` | `$C5` | 2 |
| Zero Page,X | `CMP $nn,X` | `$D5` | 2 |
| (Indirect,X) | `CMP ($nn,X)` | `$C1` | 2 |
| (Indirect),Y | `CMP ($nn),Y` | `$D1` | 2 |
| Absolute | `CMP $nnnn` | `$CD` | 3 |
| Absolute,X | `CMP $nnnn,X` | `$DD` | 3 |
| Absolute,Y | `CMP $nnnn,Y` | `$D9` | 3 |

### CPX — Compare X

Computes X - M to set C, Z and N, leaving X unchanged.

**Flags:** `N – – – – – Z C`

| Mode | Syntax | Opcode | Bytes |
|---|---|---|---|
| Immediate | `CPX #$nn` | `$E0` | 2 |
| Zero Page | `CPX $nn` | `$E4` | 2 |
| Absolute | `CPX $nnnn` | `$EC` | 3 |

### CPY — Compare Y

Computes Y - M to set C, Z and N, leaving Y unchanged.

**Flags:** `N – – – – – Z C`

| Mode | Syntax | Opcode | Bytes |
|---|---|---|---|
| Immediate | `CPY #$nn` | `$C0` | 2 |
| Zero Page | `CPY $nn` | `$C4` | 2 |
| Absolute | `CPY $nnnn` | `$CC` | 3 |

## Branches

### BPL — Branch if Plus

Branches when N = 0 (result positive). The operand is a signed offset (-128..+127) from the next instruction.

**Flags:** `– – – – – – – –`

| Mode | Syntax | Opcode | Bytes |
|---|---|---|---|
| Relative | `BPL $nnnn (label)` | `$10` | 2 |

### BMI — Branch if Minus

Branches when N = 1 (result negative). Signed-offset relative branch.

**Flags:** `– – – – – – – –`

| Mode | Syntax | Opcode | Bytes |
|---|---|---|---|
| Relative | `BMI $nnnn (label)` | `$30` | 2 |

### BVC — Branch if Overflow Clear

Branches when V = 0. Signed-offset relative branch.

**Flags:** `– – – – – – – –`

| Mode | Syntax | Opcode | Bytes |
|---|---|---|---|
| Relative | `BVC $nnnn (label)` | `$50` | 2 |

### BVS — Branch if Overflow Set

Branches when V = 1. Signed-offset relative branch.

**Flags:** `– – – – – – – –`

| Mode | Syntax | Opcode | Bytes |
|---|---|---|---|
| Relative | `BVS $nnnn (label)` | `$70` | 2 |

### BCC — Branch if Carry Clear

Branches when C = 0 (e.g. after CMP, when A < M). Signed-offset relative branch.

**Flags:** `– – – – – – – –`

| Mode | Syntax | Opcode | Bytes |
|---|---|---|---|
| Relative | `BCC $nnnn (label)` | `$90` | 2 |

### BCS — Branch if Carry Set

Branches when C = 1 (e.g. after CMP, when A >= M). Signed-offset relative branch.

**Flags:** `– – – – – – – –`

| Mode | Syntax | Opcode | Bytes |
|---|---|---|---|
| Relative | `BCS $nnnn (label)` | `$B0` | 2 |

### BNE — Branch if Not Equal

Branches when Z = 0 (last result non-zero). The workhorse loop branch.

**Flags:** `– – – – – – – –`

| Mode | Syntax | Opcode | Bytes |
|---|---|---|---|
| Relative | `BNE $nnnn (label)` | `$D0` | 2 |

### BEQ — Branch if Equal

Branches when Z = 1 (last result zero). Signed-offset relative branch.

**Flags:** `– – – – – – – –`

| Mode | Syntax | Opcode | Bytes |
|---|---|---|---|
| Relative | `BEQ $nnnn (label)` | `$F0` | 2 |

## Jump / Subroutine

### JMP — Jump

Sets the program counter to the target address. Absolute jumps to a fixed address; indirect (JMP ($xxxx)) jumps through a pointer. Note: this 6502 faithfully reproduces the indirect page-boundary bug at $xxFF.

**Flags:** `– – – – – – – –`

| Mode | Syntax | Opcode | Bytes |
|---|---|---|---|
| Absolute | `JMP $nnnn` | `$4C` | 3 |
| Indirect | `JMP ($nnnn)` | `$6C` | 3 |

### JSR — Jump to Subroutine

Pushes the return address (minus one) onto the stack, then jumps to the target. Paired with RTS to call reusable routines.

**Flags:** `– – – – – – – –`

| Mode | Syntax | Opcode | Bytes |
|---|---|---|---|
| Absolute | `JSR $nnnn` | `$20` | 3 |

### RTS — Return from Subroutine

Pulls the return address from the stack and resumes after the matching JSR.

**Flags:** `– – – – – – – –`

| Mode | Syntax | Opcode | Bytes |
|---|---|---|---|
| Implied | `RTS` | `$60` | 1 |

## Interrupt

### BRK — Force Interrupt

Forces a software interrupt: pushes PC+2 and the status (with B set), sets I, and jumps through the IRQ vector at $FFFE. RTI returns.

**Flags:** `– – – 1 – – I –`

| Mode | Syntax | Opcode | Bytes |
|---|---|---|---|
| Implied | `BRK` | `$00` | 1 |

### RTI — Return from Interrupt

Pulls the status register and the program counter from the stack, resuming the interrupted code. Restores all flags.

**Flags:** `N V x x D I Z C`

| Mode | Syntax | Opcode | Bytes |
|---|---|---|---|
| Implied | `RTI` | `$40` | 1 |

## Flags

### CLC — Clear Carry

Clears the carry flag. Do this before the first ADC of a multi-byte addition.

**Flags:** `– – – – – – – 0`

| Mode | Syntax | Opcode | Bytes |
|---|---|---|---|
| Implied | `CLC` | `$18` | 1 |

### SEC — Set Carry

Sets the carry flag. Do this before the first SBC of a subtraction.

**Flags:** `– – – – – – – 1`

| Mode | Syntax | Opcode | Bytes |
|---|---|---|---|
| Implied | `SEC` | `$38` | 1 |

### CLI — Clear Interrupt Disable

Clears the I flag, enabling maskable IRQ interrupts.

**Flags:** `– – – – – 0 – –`

| Mode | Syntax | Opcode | Bytes |
|---|---|---|---|
| Implied | `CLI` | `$58` | 1 |

### SEI — Set Interrupt Disable

Sets the I flag, masking IRQ interrupts.

**Flags:** `– – – – – 1 – –`

| Mode | Syntax | Opcode | Bytes |
|---|---|---|---|
| Implied | `SEI` | `$78` | 1 |

### CLV — Clear Overflow

Clears the V (overflow) flag. There is no SEV.

**Flags:** `– 0 – – – – – –`

| Mode | Syntax | Opcode | Bytes |
|---|---|---|---|
| Implied | `CLV` | `$B8` | 1 |

### CLD — Clear Decimal

Clears the D flag, selecting binary arithmetic for ADC/SBC.

**Flags:** `– – – – 0 – – –`

| Mode | Syntax | Opcode | Bytes |
|---|---|---|---|
| Implied | `CLD` | `$D8` | 1 |

### SED — Set Decimal

Sets the D flag, selecting binary-coded-decimal arithmetic for ADC/SBC.

**Flags:** `– – – – 1 – – –`

| Mode | Syntax | Opcode | Bytes |
|---|---|---|---|
| Implied | `SED` | `$F8` | 1 |

### NOP — No Operation

Does nothing for one instruction. Used for timing and as a placeholder.

**Flags:** `– – – – – – – –`

| Mode | Syntax | Opcode | Bytes |
|---|---|---|---|
| Implied | `NOP` | `$EA` | 1 |

## Appendix — Example Programs

Eight complete programs you can run on the Logic Bench 6502 **right now**. Every one was assembled with the bench's own assembler and run on its verified emulator, so each is known to work — and each draws to the **32×32 screen** at `$8000`, the one piece of output hardware the 6502 boards carry.

**To run one:** load **6502 — Gate-Built** (or **6502 CPU (behavioral)**) from the library, open the **ASM** panel, **copy** a listing below, paste it in, press **LOAD**, then **RUN**. The static demos (1–7) draw a picture and stop; **Game of Life** (8) animates forever — raise the clock to watch it evolve.

> **A note on input.** The screen boards (behavioral, gate, ball) expose a screen but no program-readable buttons, so these eight programs are self-contained: they compute and draw, without waiting for input. If you want a 6502 that *reads* input, load the **6502 — Pocket Calculator** board: it runs calculator firmware on the same gate-true 6502, reading a 4×4 keypad and driving a 6-digit display through the memory-mapped `$D0xx` I/O block (see the memory map). These demos are ordered easiest-first — start with the Border Frame and work down.

### Border Frame

The simplest demo: walk the edges of the screen and light each one. A gentle introduction to the `plot` subroutine and nested loops.

```
; ============================================================
; BORDER FRAME — draw a one-pixel border around the 32x32 screen
; ============================================================
.org $0200
        ; top row (y=0) and bottom row (y=31)
        LDA #$00
        STA $20          ; x = 0
top:    LDA #$00
        STA $21
        JSR plot         ; (x, 0)
        LDA #$1F
        STA $21
        JSR plot         ; (x, 31)
        INC $20
        LDA $20
        CMP #$20
        BNE top
        ; left column (x=0) and right column (x=31)
        LDA #$00
        STA $21          ; y = 0
side:   LDA #$00
        STA $20
        JSR plot         ; (0, y)
        LDA #$1F
        STA $20
        JSR plot         ; (31, y)
        INC $21
        LDA $21
        CMP #$20
        BNE side
done:   JMP done

; --- plot: light pixel at (col $20, row $21) on the 32x32 screen ---
plot:   LDA $21          ; Y index = row*4 + (col>>3)
        ASL A
        ASL A
        STA $22
        LDA $20
        LSR A
        LSR A
        LSR A
        CLC
        ADC $22
        TAY
        LDA $20          ; bit mask = $80 >> (col & 7)
        AND #$07
        TAX
        LDA #$80
psh:    CPX #$00
        BEQ pdn
        LSR A
        DEX
        JMP psh
pdn:    ORA $8000,Y      ; OR the bit in, keep the rest of the byte
```

```
        STA $8000,Y
        RTS
```

### Sierpinski Triangle

A fractal that falls straight out of bitwise logic: light pixel (x,y) whenever `x AND y` equals zero. No special math — just the AND from Part I, applied to coordinates.

```
; ============================================================
; SIERPINSKI TRIANGLE — light pixel (x,y) when (x AND y) == 0
; A fractal falls straight out of bitwise AND.
; ============================================================
.org $0200
        LDA #$00
        STA $21          ; y = 0
yloop:  LDA #$00
        STA $20          ; x = 0
xloop:  LDA $20          ; if (x AND y) == 0, plot
        AND $21
        BNE skip
        JSR plot
skip:   INC $20
        LDA $20
        CMP #$20
        BNE xloop
        INC $21
        LDA $21
        CMP #$20
        BNE yloop
done:   JMP done

; --- plot: light pixel at (col $20, row $21) on the 32x32 screen ---
plot:   LDA $21          ; Y index = row*4 + (col>>3)
        ASL A
        ASL A
        STA $22
        LDA $20
        LSR A
        LSR A
        LSR A
        CLC
        ADC $22
        TAY
        LDA $20          ; bit mask = $80 >> (col & 7)
        AND #$07
        TAX
        LDA #$80
psh:    CPX #$00
        BEQ pdn
        LSR A
        DEX
        JMP psh
pdn:    ORA $8000,Y      ; OR the bit in, keep the rest of the byte
        STA $8000,Y
        RTS
```

## Checkerboard

A classic checkerboard from a one-bit test: plot a pixel when `(x + y)` is even. Shows how a single bit of an arithmetic result can drive a pattern.

```
; ============================================================
; CHECKERBOARD — plot (x,y) when (x + y) is even
; ============================================================
.org $0200
        LDA #$00
        STA $21          ; y = 0
yloop:  LDA #$00
        STA $20          ; x = 0
xloop:  LDA $20          ; if ((x + y) AND 1) == 0, plot
        CLC
        ADC $21
        AND #$01
        BNE skip
        JSR plot
skip:   INC $20
        LDA $20
        CMP #$20
        BNE xloop
        INC $21
        LDA $21
        CMP #$20
        BNE yloop
done:   JMP done

; --- plot: light pixel at (col $20, row $21) on the 32x32 screen ---
plot:   LDA $21          ; Y index = row*4 + (col>>3)
        ASL A
        ASL A
        STA $22
        LDA $20
        LSR A
        LSR A
        LSR A
        CLC
        ADC $22
        TAY
        LDA $20          ; bit mask = $80 >> (col & 7)
        AND #$07
        TAX
        LDA #$80
psh:    CPX #$00
        BEQ pdn
        LSR A
        DEX
        JMP psh
pdn:    ORA $8000,Y      ; OR the bit in, keep the rest of the byte
        STA $8000,Y
        RTS
```

## Diagonal Cross

Both diagonals of the screen, forming an X. The second diagonal uses subtraction (31 minus i) to mirror the first — a first taste of computed coordinates.

```
; ============================================================
; DIAGONAL CROSS — both diagonals of the screen (an X)
; ============================================================
.org $0200
        LDA #$00
        STA $20          ; i = 0..31
loop:   LDA $20          ; plot (i, i)
        STA $21
        JSR plot
        LDA #$1F          ; plot (i, 31 - i)
        SEC
        SBC $20
        STA $21
        LDA $20
        JSR plot
        INC $20
        LDA $20
        CMP #$20
        BNE loop
done:   JMP done

; --- plot: light pixel at (col $20, row $21) on the 32x32 screen ---
plot:   LDA $21          ; Y index = row*4 + (col>>3)
        ASL A
        ASL A
        STA $22
        LDA $20
        LSR A
        LSR A
        LSR A
        CLC
        ADC $22
        TAY
        LDA $20          ; bit mask = $80 >> (col & 7)
        AND #$07
        TAX
        LDA #$80
psh:    CPX #$00
        BEQ pdn
        LSR A
        DEX
        JMP psh
pdn:    ORA $8000,Y      ; OR the bit in, keep the rest of the byte
        STA $8000,Y
        RTS
```

## Sine Wave

Plot a sine curve from a 32-entry lookup table. Demonstrates `.byte` data tables and indexed addressing (`LDA sintab,X`) — the 6502 way to ship precomputed data.

```
; ============================================================
; SINE WAVE — plot a sine curve from a 32-entry lookup table
; y = sintab[x], for x = 0..31
; ============================================================
.org $0200
        LDA #$00
        STA $20          ; x = 0
loop:   LDX $20
        LDA sintab,X     ; y = sintab[x]
        STA $21
        JSR plot
        INC $20
        LDA $20
        CMP #$20
        BNE loop
done:   JMP done

; --- plot: light pixel at (col $20, row $21) on the 32x32 screen ---
plot:   LDA $21          ; Y index = row*4 + (col>>3)
        ASL A
        ASL A
        STA $22
        LDA $20
        LSR A
        LSR A
        LSR A
        CLC
        ADC $22
        TAY
        LDA $20          ; bit mask = $80 >> (col & 7)
        AND #$07
        TAX
        LDA #$80
psh:    CPX #$00
        BEQ pdn
        LSR A
        DEX
        JMP psh
pdn:    ORA $8000,Y      ; OR the bit in, keep the rest of the byte
        STA $8000,Y
        RTS
; one period of a sine, scaled to rows 0..31
sintab: .byte 16,19,22,25,27,29,30,31,31,31,30,29,27,25,22,19
        .byte 16,12,9,6,4,2,1,0,0,0,1,2,4,6,9,12
```

### Concentric Rings

Nested square outlines, each inset by four pixels. Builds on the border demo with an outer loop over the inset and `BCS` range checks.

```
; ============================================================
; CONCENTRIC RINGS — nested square outlines, inset by 4 each time
; ============================================================
.org $0200
        LDA #$00
        STA $23          ; r = inset (0, 4, 8, 12)
rloop:  ; horizontal edges: for x = r..31-r, plot (x,r) and (x,31-r)
        LDA $23
        STA $20
hloop:  LDA $23
        STA $21
        JSR plot         ; (x, r)
        LDA #$1F
        SEC
        SBC $23
        STA $21
        JSR plot         ; (x, 31-r)
        INC $20
        LDA #$1F
        SEC
        SBC $23
        CMP $20
        BCS hloop
        ; vertical edges: for y = r..31-r, plot (r,y) and (31-r,y)
        LDA $23
        STA $21
vloop:  LDA $23
        STA $20
        JSR plot         ; (r, y)
        LDA #$1F
        SEC
        SBC $23
        STA $20
        JSR plot         ; (31-r, y)
        INC $21
        LDA #$1F
        SEC
        SBC $23
        CMP $21
        BCS vloop
        LDA $23          ; r += 4
        CLC
        ADC #$04
        STA $23
        CMP #$10
        BCC rloop
done:   JMP done

; --- plot: light pixel at (col $20, row $21) on the 32x32 screen ---
plot:   LDA $21          ; Y index = row*4 + (col>>3)
        ASL A
        ASL A
        STA $22
        LDA $20
```

```
        LSR A
        LSR A
        LSR A
        CLC
        ADC $22
        TAY
        LDA $20          ; bit mask = $80 >> (col & 7)
        AND #$07
        TAX
        LDA #$80
psh:    CPX #$00
        BEQ pdn
        LSR A
        DEX
        JMP psh
pdn:    ORA $8000,Y      ; OR the bit in, keep the rest of the byte
        STA $8000,Y
        RTS
```

## Moiré (multiplication texture)

An interference texture from a multiplication table: plot (x,y) when bit 3 of `x * y` is set. Since the 6502 has no multiply instruction, the product is built by repeated addition — and the bit pattern produces a shimmering moiré.

```
; ============================================================
; MOIRE — plot (x,y) when bit 3 of (x * y) is set.
; The product is computed by repeated addition (no MUL on the 6502),
; and the interference of the bit pattern makes a moire texture.
; ============================================================
.org $0200
        LDA #$00
        STA $21          ; y = 0
yloop:  LDA #$00
        STA $20          ; x = 0
xloop:  LDA #$00          ; product = x * y  (add x, y times)
        STA $24
        LDX $21
        BEQ pzero
padd:   CLC
        LDA $24
        ADC $20
        STA $24
        DEX
        BNE padd
pzero:  LDA $24          ; test bit 3 of the product
        AND #$08
        BEQ skip
        JSR plot
skip:   INC $20
        LDA $20
        CMP #$20
        BNE xloop
        INC $21
        LDA $21
        CMP #$20
        BNE yloop
done:   JMP done

; --- plot: light pixel at (col $20, row $21) on the 32x32 screen ---
plot:   LDA $21          ; Y index = row*4 + (col>>3)
        ASL A
        ASL A
        STA $22
        LDA $20
        LSR A
        LSR A
        LSR A
        CLC
        ADC $22
        TAY
        LDA $20          ; bit mask = $80 >> (col & 7)
        AND #$07
        TAX
        LDA #$80
psh:    CPX #$00
        BEQ pdn
        LSR A
        DEX
```

```
        JMP psh
pdn:    ORA $8000,Y      ; OR the bit in, keep the rest of the byte
        STA $8000,Y
        RTS
```

### Conway's Game of Life

Conway's Game of Life on the full 32x32 grid, running generation after generation. The big one: a double-buffered board, eight-neighbour counting, and the survival rules. Watch for the `BEQ` + `JMP` pairs — a 6502 branch only reaches 127 bytes either way, so the long loops are written as "branch over a JMP" (exactly the limit taught in Workbook Lesson 59).

```
; ============================================================
; CONWAY'S GAME OF LIFE — 32x32, interior cells (edges stay dead).
; Board A at $1000 (1 byte/cell), next generation in board B at $1400.
; Each generation: count 8 neighbours, apply the rules, blit to the
; screen, copy B over A, repeat forever. Seeded with a small pattern.
; Note the BEQ+JMP pairs: a 6502 branch only reaches +/-127 bytes, so
; long loops are written as "branch over a JMP" (see Workbook L59).
; ============================================================
.org $0200
        ; clear board A and the screen
        LDA #$00
        TAX
clr:    STA $1000,X
        STA $1100,X
        STA $1200,X
        STA $1300,X
        STA $8000,X
        INX
        BNE clr
        ; seed a small pattern near the top-left
        LDA #$01
        STA $1022
        STA $1043
        STA $1041
        STA $1062
        STA $1063
gen:    LDA #$01
        STA $30          ; y = 1
ny:     LDA #$01
        STA $31          ; x = 1
nx:     LDA $30          ; base = $1000 + y*32 + x  -> ptr $32/$33
        ASL A
        ASL A
        ASL A
        ASL A
        ASL A
        STA $32
        LDA #$00
        ROL A
        STA $33
        LDA $32
        CLC
        ADC $31
        STA $32
        LDA $33
        ADC #$00
        STA $33
        LDA $33
        CLC
        ADC #$10
        STA $33
        LDA #$00         ; count live neighbours
        STA $34
        LDX #$00
```

```
ncl:    LDA offlo,X
        CLC
        ADC $32
        STA $36
        LDA $33
        ADC offhi,X
        STA $37
        LDY #$00
        LDA ($36),Y
        BEQ ncz
        INC $34
ncz:    INX
        CPX #$08
        BNE ncl
        LDY #$00          ; apply the rules to the current cell
        LDA ($32),Y
        BNE alive
        LDA $34           ; dead: born on exactly 3 neighbours
        CMP #$03
        BNE dead
        JMP live
alive:  LDA $34           ; alive: survives on 2 or 3 neighbours
        CMP #$02
        BEQ live
        CMP #$03
        BEQ live
dead:   LDA #$00
        JMP store
live:   LDA #$01
store:  PHA               ; write result into board B ($1400 + offset)
        LDA $33
        CLC
        ADC #$04
        STA $39
        LDA $32
        STA $38
        PLA
        LDY #$00
        STA ($38),Y
        INC $31
        LDA $31
        CMP #$1F
        BEQ nxd
        JMP nx
nxd:    INC $30
        LDA $30
        CMP #$1F
        BEQ nyd
        JMP ny
nyd:    LDX #$00          ; clear the screen
        LDA #$00
bcs:    STA $8000,X
        INX
        CPX #$80
```

```
        BNE bcs
        LDA #$00          ; blit board B to the screen
        STA $30
by:     LDA #$00
        STA $31
bx:     LDA $30
        ASL A
        ASL A
        ASL A
        ASL A
        ASL A
        STA $32
        LDA #$00
        ROL A
        CLC
        ADC #$14
        STA $33
        LDA $32
        CLC
        ADC $31
        STA $32
        LDA $33
        ADC #$00
        STA $33
        LDY #$00
        LDA ($32),Y
        BEQ bnx
        LDA $30
        ASL A
        ASL A
        STA $3A
        LDA $31
        LSR A
        LSR A
        LSR A
        CLC
        ADC $3A
        TAY
        LDA $31
        AND #$07
        TAX
        LDA #$80
bsh:    CPX #$00
        BEQ bdn
        LSR A
        DEX
        JMP bsh
bdn:    ORA $8000,Y
        STA $8000,Y
bnx:    INC $31
        LDA $31
        CMP #$20
        BEQ bxd
        JMP bx
```

```
bxd:    INC $30
        LDA $30
        CMP #$20
        BEQ byd
        JMP by
byd:    LDX #$00          ; copy board B over board A
cpb:    LDA $1400,X
        STA $1000,X
        LDA $1500,X
        STA $1100,X
        LDA $1600,X
        STA $1200,X
        LDA $1700,X
        STA $1300,X
        INX
        BNE cpb
        JMP gen
; signed neighbour offsets: -33,-32,-31,-1,+1,+31,+32,+33
offlo:  .byte $DF,$E0,$E1,$FF,$01,$1F,$20,$21
offhi:  .byte $FF,$FF,$FF,$FF,$00,$00,$00,$00
```

---

*This manual is generated from Logic Bench's own instruction-set table and verified against its gold-oracle emulator, so it always matches the machine you can run on the bench. Load 6502 — Gate-Built from the library, open the ASM panel, and try any instruction here — then drill into the chip and watch it execute on real gates.*

*Every box opens. Nothing inside is fake.*