

# Appendix 4: Instruction Set Architecture

## ISA Overview

The SD-8516 instruction set is organized into four tiers. Numbering is canonical in decimal (as defined in `arch.h` and `arch.ts`); hex is given alongside for convenience.

- **Tier 1 - Core** behaves like a RISC instruction set: small, orthogonal, and complete on its own. Target this first. Everything above it can be synthesized from the core if you have to.
- **Tier 2 - Extended** are quality-of-life instructions for hand assemblers. `ADD reg, imm` is the canonical example: you do not need it (load the immediate and `ADD`), but it is provided for convenience. `MUL` is in the same spirit, since you can loop with `ADD`.
- **Tier 3 - CISC** are heavier, mostly VAX- and 680×0-flavored instructions: block memory, queue management, character scanning, jump tables. Each one folds a small loop into a single fetch-execute.
- **Tier 4 - Acceleration** are instructions added primarily to speed up a specific consumer: the LLVM backend (`MOVXS/MOVZX`, indexed addressing), the Forth system (`LSTEP` and friends), or the hardware (PPU/ASU). They exist to remove instructions from a hot path, not to add expressiveness.

A handful of opcodes carry aliases (e.g. `JC/JAE`, `BC/BAE`, `CMPC/CMPC3`); both names assemble to the same byte.

*Deprecated since the previous revision: `CASE3`, `CASEB`, `PAB`, `UAB`, and conditional `JMPR`.*

## Tier 1: Core (RISC)

A small, sufficient core. If you only ever target these, you can write any program.

| #  | hex  | Mnemonic                | Example                 | Description                                     | Flags |
|----|------|-------------------------|-------------------------|-------------------------------------------------|-------|
| 0  | \$00 | <a href="#">LD_IMM</a>  | <code>LDA \$5</code>    | Load register with an immediate value           |       |
| 1  | \$01 | <a href="#">LD_MEM</a>  | <code>LDA [\$5]</code>  | Load register from memory (absolute)            |       |
| 2  | \$02 | <a href="#">LD_REG</a>  | <code>LDA [X]</code>    | Load register from memory (register pointer)    |       |
| 3  | \$03 | <a href="#">LD_IMM8</a> | <code>LDAL \$5</code>   | Load byte register with an immediate            |       |
| 4  | \$04 | <a href="#">LD_MEM8</a> | <code>LDAL [\$5]</code> | Load byte from memory (absolute)                |       |
| 5  | \$05 | <a href="#">LD_REG8</a> | <code>LDAL [X]</code>   | Load byte from memory (register pointer)        |       |
| 6  | \$06 | <a href="#">ST_MEM</a>  | <code>STA [\$10]</code> | Store register to memory (absolute)             |       |
| 7  | \$07 | <a href="#">ST_REG</a>  | <code>STA [X]</code>    | Store register to memory (register pointer)     |       |
| 9  | \$09 | <a href="#">MOV</a>     | <code>MOV Y, A</code>   | Copy register to register ( <i>same width</i> ) |       |
| 11 | \$0B | <a href="#">PUSH</a>    | <code>PUSH A</code>     | Push register onto stack                        |       |
| 12 | \$0C | <a href="#">POP</a>     | <code>POP Y</code>      | Pop stack into register                         |       |
| 15 | \$0F | <a href="#">PUSHF</a>   | <code>PUSHF</code>      | Push flags                                      |       |
| 16 | \$10 | <a href="#">POPF</a>    | <code>POPF</code>       | Pop flags                                       | *     |
| 23 | \$17 | <a href="#">INC</a>     | <code>INC X</code>      | Increment register by 1 (any width)             |       |
| 24 | \$18 | <a href="#">DEC</a>     | <code>DEC Y</code>      | Decrement register by 1 (any width)             |       |

|     |      |                |              |                                          |         |
|-----|------|----------------|--------------|------------------------------------------|---------|
| 30  | \$1E | <b>ADD</b>     | ADD X, Y     | $X = X + Y$                              |         |
| 31  | \$1F | <b>SUB</b>     | SUB X, Y     | $X = X - Y$                              |         |
| 50  | \$32 | <b>AND</b>     | AND dst, src | Bitwise AND                              |         |
| 51  | \$33 | <b>OR</b>      | OR dst, src  | Bitwise OR                               |         |
| 52  | \$34 | <b>XOR</b>     | XOR dst, src | Bitwise XOR                              |         |
| 53  | \$35 | <b>NOT</b>     | NOT reg      | Bitwise NOT (invert all bits)            |         |
| 70  | \$46 | <b>SHL</b>     | SHL A        | Shift left                               | Z N C   |
| 71  | \$47 | <b>SHR</b>     | SHR A        | Shift right                              | Z N C   |
| 90  | \$5A | <b>CMP</b>     | CMP A, B     | Compare (subtract, discard result)       | Z N C V |
| 91  | \$5B | <b>CMP_IMM</b> | CMP A, \$1   | Compare against immediate                | Z N C V |
| 100 | \$64 | <b>JMP</b>     | JMP @label   | Unconditional jump                       |         |
| 101 | \$65 | <b>JZ</b>      | JZ @label    | Jump if zero                             |         |
| 102 | \$66 | <b>JNZ</b>     | JNZ @label   | Jump if not zero                         |         |
| 105 | \$69 | <b>JC</b>      | JC @label    | Jump if carry set (alias <i>JAE</i> )    |         |
| 106 | \$6A | <b>JNC</b>     | JNC @label   | Jump if carry clear (alias <i>JNAE</i> ) |         |
| 130 | \$82 | <b>CALL</b>    | CALL @label  | Call subroutine (push IP, jump)          |         |
| 133 | \$85 | <b>RET</b>     | RET          | Return from subroutine (pop IP)          |         |
| 134 | \$86 | <b>INT</b>     | INT \$10     | Software interrupt                       |         |
| 135 | \$87 | <b>RTI</b>     | RTI          | Return from interrupt                    | *       |
| 182 | \$B6 | <b>SETF</b>    | SETF \$80    | Set bits in the flags register           | *       |
| 183 | \$B7 | <b>CLRF</b>    | CLRF \$80    | Clear bits in the flags register         | *       |
| 184 | \$B8 | <b>TESTF</b>   | TESTF \$80   | Non-destructive AND against flags        | Z C     |
| 254 | \$FE | <b>NOP</b>     | NOP          | No operation                             |         |
| 255 | \$FF | <b>HALT</b>    | HALT         | Halt CPU (sets HALT flag)                |         |

## Tier 2: Extended (Quality of Life)

Conveniences for the hand assembler. None of these add capability the core lacks; they trade an opcode for fewer instructions in common patterns.

| #  | hex  | Mnemonic     | Example       | Description                               | Flags |
|----|------|--------------|---------------|-------------------------------------------|-------|
| 10 | \$0A | <b>XCHG</b>  | XCHG X, Y     | Swap two registers                        |       |
| 13 | \$0D | <b>PUSHA</b> | PUSHA         | Push all registers                        |       |
| 14 | \$0E | <b>POPA</b>  | POPA          | Pop all registers                         | *     |
| 17 | \$11 | <b>PUSH2</b> | PUSH2 A, B    | Push 2 registers                          |       |
| 18 | \$12 | <b>POP2</b>  | POP2 A, B     | Pop 2 registers                           |       |
| 19 | \$13 | <b>PUSH3</b> | PUSH3 A, B, X | Push 3 registers                          |       |
| 20 | \$14 | <b>POP3</b>  | POP3 A, B, X  | Pop 3 registers                           |       |
| 21 | \$15 | <b>PUSH4</b> | PUSH4 A,B,X,Y | Push 4 registers                          |       |
| 22 | \$16 | <b>POP4</b>  | POP4 A,B,X,Y  | Pop 4 registers                           |       |
| 25 | \$19 | <b>ST_PD</b> | STA [-, BLX]  | Pre-decrement store (alias <i>DPUSH</i> ) |       |
| 26 | \$1A | <b>LD_FS</b> | LDA [BLX, +]  | Load + forward step (alias <i>DPOP</i> )  |       |
| 27 | \$1B | <b>ST_FS</b> | STA [BLX, +]  | Store + forward step                      |       |
| 32 | \$20 | <b>MUL</b>   | MUL X, Y      | $X = X * Y$                               |       |
| 33 | \$21 | <b>DIV</b>   | DIV X, Y      | $X = X / Y$ (unsigned)                    |       |
| 34 | \$22 | <b>MOD</b>   | MOD X, Y      | $X = X \% Y$ (unsigned)                   |       |

|     |      |              |               |                                                   |       |
|-----|------|--------------|---------------|---------------------------------------------------|-------|
| 35  | \$23 | ADD_REG_IMM  | ADD X, \$1234 | X = X + immediate                                 |       |
| 36  | \$24 | SUB_REG_IMM  | SUB X, \$ABCD | X = X - immediate                                 |       |
| 37  | \$25 | MUL_REG_IMM  | MUL X, \$100  | X = X * immediate                                 |       |
| 38  | \$26 | DIV_REG_IMM  | DIV X, \$10   | X = X / immediate (unsigned)                      |       |
| 39  | \$27 | MOD_REG_IMM  | MOD X, \$FF   | X = X % immediate (unsigned)                      |       |
| 40  | \$28 | ADDC         | ADDC X, Y     | X = X + Y + carry                                 |       |
| 41  | \$29 | SUBC         | SUBC X, Y     | X = X - Y - borrow                                |       |
| 42  | \$2A | ADDC_REG_IMM | ADDC X, \$5   | X = X + immediate + carry                         |       |
| 43  | \$2B | SUBC_REG_IMM | SUBC X, \$1   | X = X - immediate - borrow                        |       |
| 54  | \$36 | TEST         | TEST dst, src | Non-destructive AND                               |       |
| 55  | \$37 | AND_IMM      | AND dst, #imm | Bitwise AND with immediate                        |       |
| 56  | \$38 | OR_IMM       | OR dst, #imm  | Bitwise OR with immediate                         |       |
| 57  | \$39 | XOR_IMM      | XOR dst, #imm | Bitwise XOR with immediate                        |       |
| 72  | \$48 | SHLC         | SHLC A        | Shift left through carry                          | Z N C |
| 73  | \$49 | SHRC         | SHRC A        | Shift right through carry                         | Z N C |
| 74  | \$4A | ROL          | ROL A         | Rotate left                                       | Z N C |
| 75  | \$4B | ROR          | ROR A         | Rotate right                                      | Z N C |
| 76  | \$4C | ROLC         | ROLC A        | Rotate left through carry                         | Z N C |
| 77  | \$4D | RORC         | RORC A        | Rotate right through carry                        | Z N C |
| 80  | \$50 | SHL_X        | SHL_X A, C    | Shift left by count ( <i>confirm semantics</i> )  | Z N C |
| 81  | \$51 | SHR_X        | SHR_X A, C    | Shift right by count ( <i>confirm semantics</i> ) | Z N C |
| 103 | \$67 | JN           | JN @label     | Jump if negative                                  |       |
| 104 | \$68 | JNN          | JNN @label    | Jump if not negative                              |       |
| 107 | \$6B | JO           | JO @label     | Jump if overflow (alias JV)                       |       |
| 108 | \$6C | JNO          | JNO @label    | Jump if no overflow (alias JNV)                   |       |
| 110 | \$6E | JMPR         | JMPR X        | Jump to address in register                       |       |
| 111 | \$6F | JL           | JL @label     | Jump if less (signed)                             |       |
| 112 | \$70 | JLE          | JLE @label    | Jump if less or equal (signed)                    |       |
| 113 | \$71 | JG           | JG @label     | Jump if greater (signed)                          |       |
| 114 | \$72 | JGE          | JGE @label    | Jump if greater or equal (signed)                 |       |
| 115 | \$73 | JA           | JA @label     | Jump if above (unsigned)                          |       |
| 116 | \$74 | JNA          | JNA @label    | Jump if not above (unsigned)                      |       |
| 120 | \$78 | BZ           | BZ @near      | Branch if zero (2-byte relative, +/-127)          |       |
| 121 | \$79 | BNZ          | BNZ @near     | Branch if not zero                                |       |
| 122 | \$7A | BC           | BC @near      | Branch if carry (alias BAE)                       |       |
| 123 | \$7B | BNC          | BNC @near     | Branch if no carry (alias BNAE)                   |       |
| 124 | \$7C | BA           | BA @near      | Branch if above (unsigned)                        |       |
| 125 | \$7D | BNA          | BNA @near     | Branch if not above (unsigned)                    |       |
| 126 | \$7E | BG           | BG @near      | Branch if greater (signed)                        |       |
| 127 | \$7F | BGE          | BGE @near     | Branch if greater or equal (signed)               |       |
| 128 | \$80 | BL           | BL @near      | Branch if less (signed)                           |       |
| 129 | \$81 | BLE          | BLE @near     | Branch if less or equal (signed)                  |       |
| 160 | \$A0 | SEZ          | SEZ           | Set zero flag                                     | Z     |
| 161 | \$A1 | SEN          | SEN           | Set negative flag                                 | N     |
| 162 | \$A2 | SEC          | SEC           | Set carry flag                                    | C     |
| 163 | \$A3 | SEV          | SEV           | Set overflow flag                                 | V     |

|     |      |     |     |                             |   |
|-----|------|-----|-----|-----------------------------|---|
| 164 | \$A4 | SEE | SEE | Set exception/extra flag    | E |
| 165 | \$A5 | SEF | SEF | Set F (free) flag           | F |
| 166 | \$A6 | SEB | SEB | Set B (bonus) flag          | B |
| 167 | \$A7 | SEU | SEU | Set U (user) flag           | U |
| 168 | \$A8 | SED | SED | Set debug (trace) flag      | D |
| 169 | \$A9 | SEI | SEI | Set interrupt-enable flag   | I |
| 170 | \$AA | SSI | SSI | Set sound-interrupt flag    | S |
| 171 | \$AB | CLZ | CLZ | Clear zero flag             | Z |
| 172 | \$AC | CLN | CLN | Clear negative flag         | N |
| 173 | \$AD | CLC | CLC | Clear carry flag            | C |
| 174 | \$AE | CLV | CLV | Clear overflow flag         | V |
| 175 | \$AF | CLE | CLE | Clear exception/extra flag  | E |
| 176 | \$B0 | CLF | CLF | Clear F flag                | F |
| 177 | \$B1 | CLB | CLB | Clear B flag                | B |
| 178 | \$B2 | CLU | CLU | Clear U flag                | U |
| 179 | \$B3 | CLD | CLD | Clear debug flag            | D |
| 180 | \$B4 | CLI | CLI | Clear interrupt-enable flag | I |
| 181 | \$B5 | CSI | CSI | Clear sound-interrupt flag  | S |

### Tier 3: CISC (VAX-style)

Heavier instructions that fold a loop into one fetch-execute. Mostly VAX- and 680×0-inspired; useful for the kernal and for hand-written string/queue code.

| #   | hex  | Mnemonic    | Example                 | Description                                 | Flags |
|-----|------|-------------|-------------------------|---------------------------------------------|-------|
| 140 | \$8C | MEMCOPY     | MEMCOPY src, dst, n     | Copy n bytes ptr to ptr                     |       |
| 141 | \$8D | SCAN        | SCAN ELM, reg           | Scan memory for a 1-4 byte needle           | Z     |
| 142 | \$8E | CMPC        | CMPC ELM, FLD, C        | Compare characters (alias <i>CMPC3</i> )    | Z C   |
| 143 | \$8F | SKPC        | SKPC ELM, AL            | Skip characters (anti-scan)                 |       |
| 144 | \$90 | SKPC_IMM    | SKPC ELM, \$20          | Skip characters (immediate needle)          |       |
| 145 | \$91 | INSQUE      | INSQUE reg, reg         | Insert into queue                           |       |
| 146 | \$92 | INSQUE_PTR  | INSQUE reg, [reg]       | Insert into queue (pointer form)            |       |
| 147 | \$93 | REMQUE      | REMQUE reg              | Remove from queue                           | Z     |
| 148 | \$94 | SCANQUE     | SCANQUE H, N, O         | Scan a queue, matching a field (reg offset) | Z     |
| 149 | \$95 | SCANQUE_IMM | SCANQUE H, N, #O        | Scan a queue, matching a field (imm offset) | Z     |
| 200 | \$C8 | CASE        | CASE AL, #limit         | Inline jump table; CALL table[selector]     | V ER  |
| 202 | \$CA | CASETAB     | CASETAB ELM, AL, #limit | Jump table at a base address                | V ER  |
| 203 | \$CB | CVTAN       | CVTAN AL                | Convert ASCII '0'-'Z' to number 0-35        | O C   |
| 204 | \$CC | CVTNA       | CVTNA AL                | Convert number 0-35 to ASCII '0'-'Z'        | O C   |

### Tier 4: Acceleration (LLVM / Forth / Hardware)

Instructions added to remove work from a specific hot path rather than to add expressiveness. Each one has a primary consumer noted below.

| # | hex | Mnemonic | Example | Description | Consumer | Flags |
|---|-----|----------|---------|-------------|----------|-------|
|---|-----|----------|---------|-------------|----------|-------|

|     |      |                         |                 |                                               |        |   |
|-----|------|-------------------------|-----------------|-----------------------------------------------|--------|---|
| 28  | \$1C | <a href="#">MOV SX</a>  | MOV SX A, AL    | Move with sign-extend (dst wider than src)    | LLVM   |   |
| 29  | \$1D | <a href="#">MOV ZX</a>  | MOV ZX A, AL    | Move with zero-extend (dst wider than src)    | LLVM   |   |
| 210 | \$D2 | <a href="#">LD_IDXI</a> | LDA [BLX + #4]  | Indexed load, signed byte immediate -128..127 | LLVM   |   |
| 211 | \$D3 | <a href="#">LD_IDXR</a> | LDA [BLX + X]   | Indexed load, register offset                 | LLVM   |   |
| 212 | \$D4 | <a href="#">ST_IDXI</a> | STA [BLX + #4]  | Indexed store, signed byte immediate          | LLVM   |   |
| 213 | \$D5 | <a href="#">ST_IDXR</a> | STA [BLX + X]   | Indexed store, register offset                | LLVM   |   |
| 220 | \$DC | <a href="#">PPIXEL</a>  | PPIXEL X, Y, C  | PPU draw pixel                                | PPU    |   |
| 221 | \$DD | <a href="#">PLINE</a>   | PLINE           | PPU draw line                                 | PPU    |   |
| 222 | \$DE | <a href="#">PRECT</a>   | PRECT           | PPU draw rectangle                            | PPU    |   |
| 223 | \$DF | <a href="#">PRECTF</a>  | PRECTF          | PPU draw filled rectangle                     | PPU    |   |
| 224 | \$E0 | <a href="#">PCIRC</a>   | PCIRC           | PPU draw circle                               | PPU    |   |
| 225 | \$E1 | <a href="#">PCIRCF</a>  | PCIRCF          | PPU draw filled circle                        | PPU    |   |
| 226 | \$E2 | <a href="#">PCLEAR</a>  | PCLEAR          | PPU clear screen                              | PPU    |   |
| 227 | \$E3 | <a href="#">PBLIT</a>   | PBLIT           | PPU draw tile (blit)                          | PPU    |   |
| 228 | \$E4 | <a href="#">PTIMER</a>  | PTIMER          | PPU timer function                            | PPU    |   |
| 229 | \$E5 | <a href="#">PNTIMER</a> | PNTIMER         | PPU timer function (variant)                  | PPU    |   |
| 240 | \$F0 | <a href="#">LSTEPM</a>  | LSTEPM          | In-memory loop step at [FLX]                  | Forth  | Z |
| 241 | \$F1 | <a href="#">TTOS</a>    | TTOS            | Test top of stack                             | Forth  | Z |
| 242 | \$F2 | <a href="#">LIT</a>     | LIT             | Push AB, load next 4 bytes into AB            | Forth  |   |
| 243 | \$F3 | <a href="#">LSTEP</a>   | LSTEP A, @label | Decrement-and-branch (SOB / DBcc style)       | Forth  | Z |
| 251 | \$FB | CNOP                    | CNOP            | Instruction-count / timer probe               | System |   |
| 252 | \$FC | YIELD                   | YIELD           | Yield thread priority                         | System | Y |
| 253 | \$FD | BREAK                   | BREAK           | Breakpoint (reserved)                         | System |   |

## Dictionary

Detailed notes per instruction. Linkable anchors are provided for cross-reference from the tables above.

### #0 \$00 LD\_IMM

Load register with an immediate value. A = \$5 (16/24/32-bit depending on register width).

### #1 \$01 LD\_MEM

Load register from memory (absolute address). A = [\$5].

### #2 \$02 LD\_REG

Load register from memory (register-indirect pointer). A = [X].

### #3 \$03 LD\_IMM8

Load byte register with an immediate value. AL = \$5.

### #4 \$04 LD\_MEM8

Load byte from memory (absolute). AL = [\$5].

### #5 \$05 LD\_REG8

Load byte from memory (register-indirect). AL = [X].

### #6 \$06 ST\_MEM

Store register to memory (absolute). [\$10] = A.

**#7 \$07 ST\_REG**

Store register to memory (register-indirect).  $[X] = A$ .

**#9 \$09 MOV dst, src**

Register-to-register copy ex.  $Y = A$ . **MOV requires source and destination of equal width**; this is enforced at assemble time. A move that changes width is not a MOV – use [MOVSX/MOVZX](#) to widen. The assembler rejects a mismatched-width MOV rather than guessing which bytes you meant. Mismatched MOV is undefined at execution time.

**#11 \$0B PUSH**

Push register onto stack.  $SP -= \text{width}$ ;  $[SP] = A$ .

**#12 \$0C POP**

Pop stack into register.  $Y = [SP]$ ;  $SP += \text{width}$ .

**#15 \$0F PUSHF**

Push flags register onto stack.

**#16 \$10 POPF**

Pop stack into flags register. \*

**#23 \$17 INC**

Increment register by 1 (any width).  $X += 1$ . Sets Z/N flags.

**#24 \$18 DEC**

Decrement register by 1 (any width).  $Y -= 1$ . Sets Z/N flags.

**#25 ST\_PD / #26 LD\_FS / #27 ST\_FS**

Auto-stepping addressing. ST\_PD pre-decrements the pointer then stores (STA  $[-, \text{BLX}]$ ), giving a push-style write. LD\_FS loads then advances the pointer (LDA  $[\text{BLX}, +]$ ), and ST\_FS stores then advances. These collapse the pointer bookkeeping of a copy loop into the access itself.

**#28 \$1C MOVSX dst, src / #29 \$1D MOVZX dst, src**

Move with sign-extend (MOVSX) or zero-extend (MOVZX). The source is read at its own width and written across the full destination, with the high bytes filled from the source's sign bit (MOVSX) or with zero (MOVZX). The destination **must be strictly wider** than the source; this is checked at assemble time. Both fully define the destination, so unlike a byte ALU op there is no partial-register hazard, and the in-place overlapping case (MOVSX  $A, AL$ ) is well defined.

These exist to collapse the compiler's extend sequence: without a native sign-extend, widening a signed char to int took three ops (AND/XOR/SUB implementing the  $(x \wedge 0x80) - 0x80$  identity). MOVSX does it in one. Integer promotion makes signed widening pervasive in C, so this is a recurring code-size win. MOVZX similarly drops the masking immediate of an AND  $\$FF$  style zero-extend.

**#30 \$1E ADD**

Add registers.  $X = X + Y$ . Updates Z N C V.

**#31 \$1F SUB**

Subtract registers.  $X = X - Y$ . Updates Z N C V.

**#50 \$32 AND**

Bitwise AND.  $\text{dst} = \text{dst} \& \text{src}$ . Updates Z N.

**#51 \$33 OR**

Bitwise OR.  $\text{dst} = \text{dst} \mid \text{src}$ . Updates Z N.

**#52 \$34 XOR**

Bitwise XOR.  $\text{dst} = \text{dst} \wedge \text{src}$ . Updates Z N.

**#53 \$35 NOT**

Bitwise NOT (invert all bits).  $\text{reg} = \sim \text{reg}$ . Updates Z N.

**#70 \$46 SHL**

Shift left.  $A \ll= \text{count}$ . Updates Z N C.

**#71 \$47 SHR**

Shift right (logical).  $A \gg= \text{count}$ . Updates Z N C.

**#80 SHL\_X / #81 SHR\_X**

Shift by a variable count rather than by one. *The exact operand form (count in a register vs. immediate) and whether the count is masked should be confirmed against the current emulator handler before relying on this entry.*

**#90 \$64 CMP**

Compare registers (subtract, discard result).  $\text{tmp} = A - B$ . Updates Z N C V.

**#91 \$5B CMP\_IMM**

Compare register against immediate.  $\text{tmp} = A - \$1$ . Updates Z N C V.

**#100 \$64 JMP**

Unconditional jump.  $\text{IP} = @\text{label}$ .

**#101 \$65 JZ**

Jump if zero. if (Z)  $\text{IP} = @\text{label}$ .

**#102 \$66 JNZ**

Jump if not zero. if (!Z)  $\text{IP} = @\text{label}$ .

**#105 \$69 JC**

Jump if carry set (alias JAE for unsigned  $\geq$ ). if (C)  $\text{IP} = @\text{label}$ .

**#106 \$6A JNC**

Jump if carry clear (alias JNAE for unsigned  $<$ ). if (!C)  $\text{IP} = @\text{label}$ .

**#130 \$82 CALL addr / CALL reg / #131 \$83 CALLR reg**

CALL pushes the return address (the instruction after the call) and jumps to the target. CALLR takes a register operand and dispatches on its width: 16-bit and 8-bit registers combine with the current bank, 24-bit registers give a full address, and a 32-bit pair (e.g. AB) is narrowed to 24 bits by discarding the high byte. This makes 32-bit register pairs usable as function pointers.

**#120-#129 Branches (BZ, BNZ, BC, BNC, BA, BNA, BG, BGE, BL, BLE)**

Two-byte relative branches with a signed 8-bit displacement (range -128..+127 from the instruction). They mirror the absolute jumps but encode shorter and stay position-independent, so they are preferred for tight local loops. BC/BAE and BNC/BNAE are aliases. For targets outside +/-127, use the absolute jumps.

**#140 \$8C MEMCOPY src, dst, reg**

Copy reg bytes from src to dst. Handles overlap. A fast MMU operation – where a string copy is otherwise a tight LDA/STA/JNZ loop, this is a single opcode when you know the length.

**#141 \$8D SCAN ELM, reg**

Scan from ELM for the value in reg. An 8-bit register searches for a byte; a wider register searches for that many bytes. Useful for finding keyword candidates and for computing string length (scan for zero, subtract the start pointer).

**#142 \$8E CMPC ELM, FLD, C (alias CMPC3)**

Non-zero byte compare, useful for strings. Compares up to C characters; C returns either the index of the first mismatch or the matched length. Sets ZERO on a full match; otherwise CARRY distinguishes the -1 / +1 ordering.

CMPC allows early termination when `byte_a == 0` – the C-string “begins with” semantics. If both strings reach a terminator at the same position with all prior bytes equal, the loop exits matched (`Z=1`, `C=1`). Only `byte_a` is tested because by that point `byte_a == byte_b` is already proven, so `'byte_a == 0` implies `byte_b == 0`. This lets CMPC do double duty: fixed-length compare and null-terminated strcmp in one instruction.

**#143 \$8F SKPC ELM, reg / #144 \$90 SKPC\_IMM ELM, #imm**

Anti-scan: advance while the needle is found, stopping at the first non-matching position. A word-or-wider needle steps by needle width. Most often used to skip spaces: SKPC ELM, \$20 leaves ELM just past the last space.

**#145 INSQUE / #146 INSQUE\_PTR / #147 REMQUE**

Doubly-linked queue primitives. INSQUE inserts an element (register or pointer form); REMQUE removes one.

REMQUE's Z-flag semantic differs from the VAX and is worth a kernal-side note. The VAX used V for “the queue was already empty” (an error condition). SD-8516 uses Z for “the queue is now empty after this removal” (a state condition):

- VAX: V=1 means “you tried to remove from an empty queue.”
- SD-8516: Z=1 means “you removed the last item; the queue is now empty.”

Both are useful, but they answer different questions.

**#148 \$94 SCANQUE H, N, 0 / #149 \$95 SCANQUE\_IMM H, N, #0**

Three operands: head, needle, and offset. SCANQUE\_IMM takes the offset as an immediate byte; SCANQUE takes it in a register (allowing offsets beyond +255).

SCANQUE combines queue traversal with a structured-field match in one fetch-execute cycle – the inner loop of every lookup in a hash chain, free list, or PCB table. The immediate form is what you use 99% of the time (the field offset is known when you write CRUD for a struct); the register form exists so generic kernel queue helpers can take “field offset” as a parameter.

**#160 SEZ / #161 SEN / #162 SEC / #163 SEV**

Set the zero, negative, carry, and overflow flags respectively. Useful when you are certain no intervening operation disturbs the flag – for example, carry survives POP and MOV, so it is a common error/clear channel out of a routine (JC @error).

**#164 \$A4 SEE**

Set the Exception flag. Used by some interrupts and the system to signal an error, but free for your own use otherwise. Think of it as an “Extra” flag.

#### #165 \$A5 SEF

Set the F (“free”) flag. Used for “first-statement” in BASIC; otherwise safe for machine-language use.

#### #166 \$A6 SEB

Set the B (“bonus”) flag. Used for BREAK in BASIC; otherwise safe for machine-language use.

#### #167 \$A7 SEU

Set the User flag. Not used by the system; reserved for the programmer.

#### #168 \$A8 SED

Set Debug. Emits trace messages while on; slows the system considerably. Trace output may be compiled out in some builds.

#### #169 \$A9 SEI

Enable/disable interrupts. With interrupts off, INT will not fire.

#### #170 \$AA SSI

Set the Sound Interrupt flag. Managed by the KERNAL; semi-reserved and may be removed.

#### #200 \$C8 CASE selector, #limit

Inline jump table. Indexes an address from the table that follows the instruction and CALLs it. `limit` (immediate, 0-255) is the table length. If `selector > limit` the instruction silently falls through without calling; if you need to detect that, have your handlers return a result code whose default means “no handler ran.” Table format: `[addr][addr]...`

CASE reads from the instruction stream as data – the inline table is structured bytes, not fetched-and-decoded instructions, the same idea as LD\_IMM's operand pushed further. Two derived addresses do the bookkeeping:

- $IP + 3 * sel$ : the table slot holding the target (read as a 24-bit address).
- $IP + 3 * lim + 3$ : the first byte past the table (the return point, and the

IP destination on out-of-bounds).

Both are masked to 24 bits because IP arithmetic wraps in a 24-bit space.

Out-of-bounds sets V and ER and falls through rather than trapping (unlike VAX CASE, which raises an exception). The policy is “error recoverable”: software decides whether to handle it, e.g. a JV `some_handler` after the CASE. No other flags are touched, which is convenient if the selector came from an arithmetic op whose flags you still want.

A single-entry CASE acts as a CALLZ: call if the register is zero, otherwise not.

#### #202 \$CA CASETAB base, selector, #limit

Jump table held at a base address rather than inline. Indexes an address from the table at `base` by `selector`, bounded by `limit`, and dispatches to it. *The precise indexing and out-of-bounds policy should be confirmed against the current emulator handler: the older CASE3 (computed table at base) and CASEB (scanned `[selector][addr]` table) were removed, and CASETAB is the surviving table-at-base form.*

**#203 \$CB CVTAN reg8**

Convert ASCII to number. Maps '0'-'Z' to 0-35. Sets Overflow if the character is not a decimal digit (0-9) and Carry if it is not a hex digit (0-15). Also a fast digit test: MOV reg8, AL then CVTAN reg8 lets you branch with JO/JNO (JV/JNV). Designed for zoned decimal, and works for zoned hex.

**#204 \$CC CVTNA reg8**

Convert number to ASCII, the inverse of CVTAN. Maps 0-35 to '0'-'Z'. Sets Overflow if outside 0-9 and Carry if outside 0-15.

**#210 LD\_IDXI / #211 LD\_IDXR / #212 ST\_IDXI / #213 ST\_IDXR**

Indexed load/store: a 24-bit pointer register plus a displacement. The I forms take a signed byte immediate (-128..+127); the R forms take a register offset. These back the compiler's frame-slot and struct-field access ([ptr + disp]). The pointer base must be a 24-bit register; the effective address reads the base at its true width before applying the offset.

**#220-#229 PPU accelerators**

Delegate a drawing or timing primitive to the PPU: PPIXEL (pixel), PLINE (line), PRECT / PRECTF (rectangle, outline / filled), PCIRC / PCIRCF (circle, outline / filled), PCLEAR (clear screen), PBLIT (tile blit), and PTIMER / PNTIMER (timer functions). Example: PPIXEL X, Y, C draws a pixel at (X, Y) in color C.

**#240 \$F0 LSTEPM**

In-memory loop step, a Forth accelerator. Increments the 4-byte loop counter at [FLX], compares it to the 4-byte limit at [FLX+4], and sets Z when they match. This enables a very tight count-from-X-to-Y loop at the cost of an 8-byte counter/limit in memory:

```
LDFLX $F000      ; loop start
LDA #1
STA [FLX]
LDA #1000
STA [FLX+4]
loop:
LSTEPM
JNZ @loop
```

Unlike a DEC loop it counts upward and supports an arbitrary start. It is a kludge for Forth and has been generalized by [LSTEP](#); once Forth uses LSTEP, LSTEPM is a candidate for removal.

**#241 \$F1 TTOS**

Test top of stack (Forth). CD = AB; AB = [ELY]; ELY += 4; Z = (CD == 0).

**#242 \$F2 LIT**

Push literal (Forth). Pushes AB to the stack and loads the next 4 bytes of the instruction stream into AB. Encoding: [242, b0, b1, b2, b3].

**#243 \$F3 LSTEP reg, @addr**

Loop step, the generalized Forth accelerator: a decrement-and-branch in one opcode, inspired by the VAX SOB (subtract one and branch) and the 68000 DBcc.

```
LDC #1000      ; loop range
```

```
loop:  
    LSTEP C, @loop
```

From:

<https://www.appledog.ca/wiki/> - **Appledog**

Permanent link:

<https://www.appledog.ca/wiki/doku.php?id=sd:isa&rev=1781612054>

Last update: **2026/06/16 12:14**

