Coding the Impossible

Z80 Demoscene Techniques for Modern Makers

Alice Vinogradova

Alice Vinogradova — v13 (2026-02-24)

Contents

Chapter 1: Thinking in Cycles **2**
T-States: The Currency of the Z80 . . . 2
Machine Cycles and Memory Access . . . 3
The Frame: Your Canvas . . . 4
Pentagon vs. Wait Machines . . . 5
The Original Sinclair Machines . . . 5
The Pentagon: Clean Timing . . . 5
Thinking in Budgets . . . 6
Practical: Setting Up Your Development Environment . . . 7
What You Need . . . 7
Project Structure . . . 7
Build Configuration . . . 8
Practical: The Timing Harness . . . 8
Variations . . . 10
What Fits in a Frame? . . . 10
Historical Note: Dark's Advice . . . 10
The Computation Scheme . . . 11
Summary . . . 12
Try It Yourself . . . 12

Chapter 2: The Screen as a Puzzle **13**
The Memory Map: 6,912 Bytes of Screen . . . 13
The Interleave: Where the Rows Live . . . 15
Why? . . . 16
The Bit Layout: Decoding (x, y) into an Address . . . 16
The address calculation in Z80 . . . 17
Introspec's Optimisation . . . 19
The Attribute Clash . . . 21
The Border: More Than Decoration . . . 22
Border Effects . . . 23
Moving down one pixel row . . . 25
Moving down one character row (8 pixels) . . . 25
Moving up one pixel row . . . 25
Computing the attribute address from a pixel address . . . 26
Putting It Together: What the Screen Layout Means for Code . . . 27
Summary . . . 28
Try It Yourself . . . 28

Chapter 3: The Demoscener's Toolbox **30**
Unrolled Loops and Self-Modifying Code . . . 30

The cost of looping . . . 30
Unrolling: trade ROM for speed . . . 30
Self-modifying code: the Z80's secret weapon . . . 31
Self-modifying variables: the `$+1` pattern . . . 31
The Stack as a Data Pipe . . . 32
Why PUSH is the fastest write on the Z80 . . . 32
The technique . . . 33
POP as a fast read . . . 34
Where PUSH tricks are used . . . 34
LDI Chains . . . 35
LDI vs LDIR . . . 35
When LDI chains shine . . . 35
ADD A,A vs SLA A . . . 36
Code Generation . . . 36
Code generation: writing the program that draws . . . 36
Offline: generating assembly from a higher-level language . . . 36
Runtime: the program writes machine code during execution . . . 37
Method 2: Square Table Lookup . . . 42
Method 2: Logarithmic Division . . . 44
Sine and Cosine . . . 45
The Parabolic Approximation . . . 45
Trap-Based Termination . . . 47
Fixed-Point Arithmetic . . . 47
Format 8.8 . . . 47
Why Fixed-Point Matters . . . 48
Theory and Practice . . . 49
What Dark Got Right . . . 49
Random Numbers: When Tables Won't Do . . . 50
The R Register Trick . . . 50
Four Generators from the Community . . . 50
Shaped Randomness . . . 53
Seeds and Reproducibility . . . 53
Comparison Table . . . 54

Chapter 5: 3D on 3.5 MHz **56**
The Problem: Twelve Multiplications Per Vertex . . . 56
The Midpoint Method . . . 56
The Cube as Foundation . . . 57
Deriving Vertices by Averaging . . . 57
The Virtual Processor . . . 58
Architecture . . . 58
Execution . . . 58
Projection . . . 61
Parallel Projection . . . 61
Convex Polygon Filling . . . 64
The Shape of Objects . . . 66
Practical: A Spinning Solid Object . . . 66
Step 1: Define the Object . . . 67
Step 2: Write the Midpoint Program . . . 67
Step 3: Define Faces . . . 67
Step 4: The Frame Loop . . . 67

Counting T-States . . . 71
The Code Generation Pass . . . 72
What Dark Knew: Spectrum Expert and the Building Blocks . . . 72
The Hype Debate: Inner Loops vs. Mathematics . . . 73
Practical: A Simplified 56x56 Spinning Sphere . . . 73
Step 1: Precompute the Sphere Geometry . . . 73
Step 2: Build Skip Tables . . . 74
Step 3: Generate the Rendering Code . . . 74
Step 5: Source Image Layout . . . 75
The Larger Pattern . . . 75
Summary . . . 76

Chapter 7: Rotozoomer and Chunky Pixels 77
What a Rotozoomer Actually Does . . . 77
Fixed-Point Stepping on the Z80 . . . 78
Chunky Pixels: Trading Resolution for Speed . . . 78
Why 2x2 Is the Sweet Spot . . . 79
The $03 Encoding Trick . . . 79
The Inner Loop from Illusion . . . 80
Self-Modifying Code at the Byte Level . . . 80
Per-Frame Code Generation . . . 81
Buffer to Screen Transfer . . . 82
Deep Dive: 4x4 Chunky Pixels (sq, Hype 2022) . . . 82
Performance Comparison . . . 83
Historical Roots: Born Dead #05 and the Scene Lineage . . . 83
Practical: Building a Simple Rotozoomer . . . 84
Texture Design and Boundary Handling . . . 84
Why Page-Aligned, Why 256 Columns . . . 85
Choosing Texture Size . . . 85
What About Screen Boundaries? . . . 85
The Design Space . . . 86
Three Approaches to Texture Rotation . . . 86
Variant 1: Monochrome Bitmap (Full Pixel Resolution) . . . 86
Variant 2: Chunky Rotozoomer (2x2 or 4x4 Blocks) . . . 87
Variant 3: Attribute Rotozoomer (8x8 Block “Pixels”) . . . 87
Comparison . . . 88
The Rotozoomer in Context . . . 89
Summary . . . 89

Chapter 8: Multicolor — Breaking the Attribute Grid 91
The ULA’s Perspective . . . 92
The LDPUSH Insight . . . 92
LD DE,nn / PUSH DE . . . 93
How much fits in a scanline? . . . 93
The stack pointer as a cursor . . . 93
The GLUF Engine: Multicolor in a Real Game . . . 94
Double buffering . . . 94
The two-frame architecture . . . 94
What the player sees . . . 95
Ringo: A Different Kind of Multicolor . . . 95
The 11110000b pattern . . . 95
Two-screen switching . . . 96

Where the T-states go . . . 96
Horizontal scrolling . . . 97
Traditional Multicolor: The Interrupt-Driven Approach . . . 97
Step 2: Attribute changes within the display code . . . 99
Step 4: Sprite overlay . . . 100
Step 5: The main loop . . . 100
Four-Fold Symmetry: Divide and Conquer . . . 103
The Chaos Zoomer . . . 104
Code Generation: Processing Writes Z80 . . . 105
The Zapilator Question . . . 105
A Taste of the Scripting Engine . . . 106
The Making Of: Timeline and Inspiration . . . 107
Practical: Building a Simplified Attribute Tunnel . . . 107
Step 1: Fill Pixel Memory with a Pattern . . . 107
Step 2: Sine Table . . . 108
Step 3: Plasma for One Quarter . . . 108
Key Insight . . . 109
Sources . . . 110

Chapter 10: The Dotfield Scroller and 4-Phase Colour 111
Part 1: The Dotfield Scroller . . . 111
What the Viewer Sees . . . 111
The Font as Texture . . . 111
Stack-Based Address Tables . . . 112
The Inner Loop . . . 113
Beyond Simple Sine: Lissajous, Helix, and Multi-Wave Patterns . . . 115
Part 2: 4-Phase Colour Animation . . . 115
The Colour Problem . . . 115
The Trick . . . 115
The Maths of Perception . . . 116
Why Inversion Is Essential . . . 116
Practical Cost . . . 117
Text Overlay . . . 117
Demoscene Lineage . . . 118
The Shared Principle: Temporal Cheating . . . 119
Practical 1: A Bouncing Dot-Matrix Text Scroller . . . 119
Practical 2: A 4-Phase Colour Cycling Animation . . . 120
Summary . . . 120
Try It Yourself . . . 121

Chapter 11: Sound Architecture – AY, TurboSound, and Triple AY 122
11.1 The AY-3-8910 Register Map . . . 122
Complete Register Table . . . 123
Tone Channels (R0-R5): How Pitch Works . . . 123
The Noise Generator (R6) . . . 124
R7: The Mixer – The Most Important Register . . . 124
Volume Registers (R8-R10) . . . 125
Envelope Generator (R11-R13) . . . 125
11.2 Chiptune Techniques on 3 Channels . . . 126
Arpeggios: Fake Chords . . . 126
The Period Alignment Problem . . . 129
Natural Tuning: Table #5 . . . 129

Drum Synthesis . . . 132
Ornaments: Per-Frame Modulation . . . 133
11.3 TurboSound: 2 x AY . . . 134
Chip Selection . . . 134
11.4 Triple AY on ZX Spectrum Next . . . 135
Enhanced Features . . . 135
9 Channels: Orchestral Thinking . . . 135
11.5 Music Engine Architecture . . . 136
The Interrupt-Driven Player . . . 136
Frame Budget . . . 137
Formats and Trackers . . . 138
Procedural SFX Tables . . . 140
11.7 Putting It Together: The Working Example . . . 140
Sidebar: Beeper – A Brief History of Impossibility . . . 141
Sidebar: Agon Light 2 – VDP Sound System . . . 142
11.8 Practical Exercises . . . 142
Summary . . . 143

Chapter 12: Digital Drums and Music Sync 144
12.1 Digital Drums on the AY . . . 144
The Problem: The AY Cannot Play Samples . . . 144
The Cost: CPU Annihilation . . . 145
n1k-o's Insight: The Hybrid Drum . . . 145
The Buffer Dynamics . . . 148
12.3 The Scripting Engine . . . 149
Why You Need a Script . . . 149
Outer Script: The Sequence of Effects . . . 149
Inner Script: Variations Within an Effect . . . 149
kWORK: The Key Command . . . 150
How It Works in Practice . . . 152
The Threading Model . . . 152
Practical Considerations . . . 153
12.6 Practical: A Minimal Scripted Demo Engine . . . 153
What We Build . . . 153
The Memory Map . . . 153
The Timeline Script . . . 154
The Display ISR . . . 155
Observations . . . 157
12.7 Practical Exercises . . . 157
Summary . . . 158

Chapter 13: The Craft of Size-Coding 160
13.1 What is Size-Coding? . . . 160
The Z80 Size-Coder's Toolkit . . . 161
The Transposition Effect . . . 165
From Pirate Loaders to Demo Art . . . 165
Why It Matters for Size-Coding . . . 165
13.5 512-Byte Intros: Room to Breathe . . . 165
What Each Size Tier Enables . . . 166
Common 512-Byte Patterns . . . 166
Self-Modifying Tricks . . . 167
The ORG Trick . . . 167

13.6 4K Intros: The Mini-Demo . . . 167
Compression Becomes Viable . . . 167
Music Fits . . . 168
Multi-Effect Structure . . . 168
Competition Categories . . . 169
13.7 Practical: Writing a 256-Byte Intro Step by Step . . . 169
Step 1: The Unoptimised Version . . . 170
Step 2: Replace CALL with RST . . . 170
Step 3: Overlap Data with Code . . . 170
Step 4: Exploit Register State . . . 170
Step 5: Smaller Encodings Everywhere . . . 170
Step 6: Counting Bytes Precisely . . . 170
The Final Push . . . 171
13.8 Size-Coding Music: Bytebeat on AY . . . 172
The Minimal AY Formula Engine . . . 172
Techniques for Better-Sounding Formulas . . . 172
Bytebeat vs. Sequenced Music . . . 173
13.9 Procedural Graphics: The Rendered GFX Compo . . . 173
Why the Spectrum Is Interesting for This . . . 173
Common Approaches . . . 174
The Byte Budget for Art . . . 174
13.10 Size-Coding as Art . . . 174
Summary . . . 174
Try It Yourself . . . 175

Chapter 14: Compression — More Data in Less Space 177
The Memory Problem . . . 177
Compression as bandwidth amplifier . . . 178
The Benchmark . . . 178
The corpus . . . 178
The results . . . 178
The tradeoff triangle . . . 180
When to use ZX0 . . . 182
RLE and Delta Coding . . . 182
RLE: Run-Length Encoding . . . 183
Delta coding: store what changed . . . 183
The Practical Pipeline . . . 183
From asset to binary . . . 183
Makefile integration . . . 184
Example: loading screen with ZX0 . . . 184
A Practical Memory Map for a 128K Game . . . 187
15.2 Contended Memory: The Practical Truth . . . 189
What Gets Contended . . . 189
How Much Slower? . . . 189
The Practical Response . . . 190
15.3 ULA Timing . . . 190
Frame Structure . . . 190
Tact-Maps: Frame Regions . . . 191
Scanline Timing . . . 191
Total vs Practical Budget . . . 192
15.4 Floating Bus, ULA Snow, and the $7FFD Bug . . . 192

Floating Bus . . . 193
The $7FFD Read Bug . . . 193
ULA Snow . . . 193
15.5 Clone Differences . . . 193
Pentagon 128 . . . 193
Scorpion ZS-256 . . . 194
ZX Spectrum Next . . . 195
15.6 Agon Light 2: A Different Beast . . . 196
Dual-Processor Architecture . . . 196
eZ80 Memory Model: 24-Bit Flat . . . 197
ADL Mode vs Z80 Mode . . . 197
MOS API: The Operating System . . . 198
VDP Commands: Talking to the Screen . . . 198
15.7 Comparing the Platforms . . . 199
15.8 Practical: Memory Inspector Utility . . . 200
Spectrum Version . . . 200
Summary . . . 203

Chapter 16: Fast Sprites 204
Method 1: XOR Sprites . . . 204
The simplest approach . . . 204
Cycle counting . . . 207
Byte alignment and the shift problem . . . 207
Method 3: Pre-Shifted Sprites . . . 208
The memory-for-speed trade-off . . . 208
Memory calculation . . . 208
Practical compromise . . . 209
Method 4: Stack Sprites (The PUSH Method) . . . 209
The fastest output on the Z80 . . . 209
The inner loop . . . 210
The costs . . . 211
When to use stack sprites . . . 211
Method 5: Compiled Sprites . . . 211
The sprite is the code . . . 211
How it works . . . 211
The trade-offs . . . 213
Compiled sprites with masking . . . 214
Method 6: Dirty Rectangles . . . 214
The background problem . . . 214
The save/restore cycle . . . 215
Save/restore routine . . . 215
Moving a sprite . . . 219
The Algorithm: Scroll Up by One Pixel . . . 224
Cost Analysis . . . 224
Partial Scrolling: The Practical Approach . . . 224
Scrolling by 8 Pixels (One Character Row) . . . 225
Horizontal Pixel Scrolling . . . 226
Why Horizontal Scrolling Is Expensive . . . 226
The Combined Method: Character Scroll + Pixel Offset . . . 228
How It Works . . . 228
The Simple Combined Method . . . 229

Implementation: The Edge-Column Pixel Shift . . . 230
The Rendering Pipeline . . . 230
The Shadow Screen Trick . . . 231
Shadow Screen Scrolling Strategy . . . 233
Comparison: Scrolling Methods on ZX Spectrum . . . 233
Scrolling Right (and the Direction Problem) . . . 234
Agon Light 2: Hardware Scrolling . . . 235
Hardware Scroll Offsets . . . 235
Tilemap Scrolling . . . 235
Ring-Buffer Column Loading . . . 236
Agon Version: Hardware Tilemap Scrolling . . . 238
Vertical + Horizontal: Combined Scrolling . . . 239
Optimisation Tips . . . 239
1. Use a Screen Address Lookup Table . . . 239
2. Scroll Only What Is Visible . . . 239
3. Use PUSH for the Character Scroll . . . 239
4. Split the Character Scroll Across Frames . . . 239
5. Palette and Attribute Tricks . . . 240
Summary . . . 240

Chapter 18: Game Loop and Entity System 242
18.1 The Main Loop . . . 242
The Jump Table . . . 244
The Dispatcher . . . 244
Why Not a Chain of Comparisons? . . . 246
State Transitions . . . 246
18.3 Input: Reading the Player . . . 246
ZX Spectrum Keyboard . . . 246
Edge Detection: Press vs Hold . . . 248
18.4 The Entity Structure . . . 250
Structure Layout . . . 250
Why 10 Bytes? . . . 250
Why 16-bit X but 8-bit Y? . . . 251
The 8.8 Fixed-Point System . . . 251
Entity Slot Allocation . . . 252
Iterating Entities . . . 252
The Player Update Handler . . . 253
Deactivating an Entity . . . 255
Bullet Update Handler . . . 255
18.9 Agon Light 2: The Same Architecture, More Room . . . 268
18.10 Design Decisions and Trade-Offs . . . 269
Fixed vs Variable Frame Rate . . . 269
Entity Size: Lean vs Generous . . . 269
When to Use HL Instead of IX . . . 269
Summary . . . 270
Try It Yourself . . . 271

Chapter 19: Collisions, Physics, and Enemy AI 272
Part 1: Collision Detection . . . 272
AABB: The Only Shape You Need . . . 272
Ordering the Tests for Fastest Rejection . . . 275
Why Fixed-Point Matters Here . . . 279

Jump: The Anti-Gravity Impulse . . . 279
Friction: Slowing Down on the Ground . . . 280
Applying Velocity to Position . . . 281
Part 3: Enemy AI . . . 282
The Finite State Machine . . . 282
The JP Table . . . 282
Chase: The Relentless Follower . . . 284
Retreat: The Reverse Chase . . . 286
Optimisation: Update AI Every 2nd or 3rd Frame . . . 287
Part 4: Practical – Four Enemy Types . . . 288
Wiring It Into the Game Loop . . . 290
Agon Light 2 Notes . . . 291
Tuning Guide . . . 291
Summary . . . 292
Try It Yourself . . . 293

Chapter 20: Demo Workflow — From Idea to Compo 294
20.1 What "Design" Means in a Demo . . . 294
20.2 Lo-Fi Motion: A Complete Case Study . . . 295
The Concept: "Belarusian Pixel" . . . 295
The Scene Table . . . 295
Fourteen Effects . . . 296
The Effects Themselves . . . 297
The Toolchain . . . 297
The Build Pipeline . . . 297
The Timeline: Two Weeks of Evenings . . . 298
20.3 Making-of Culture . . . 299
Eager: The Technical NFO . . . 299
GABBA: A Different Workflow . . . 299
NHBF: The Puzzle . . . 300
20.4 The Toolchain in Detail . . . 300
Assembler: sjasmplus . . . 301
Emulators . . . 301
Graphics and Code Generation . . . 301
Build Automation and CI . . . 301
20.5 Compo Culture . . . 301
The Major Parties . . . 301
How to Enter Your First Compo . . . 302
20.6 The Community . . . 303
Hype (hype.retroscene.org) . . . 303
ZXArt (zxart.ee) . . . 303
Pouet (pouet.net) . . . 304
20.7 Project Management for Demo Makers . . . 304
The Minimum Viable Demo . . . 304
Working with Collaborators . . . 304
Debugging and Testing . . . 305
20.8 Transcending the Platform: Introspec's "MORE" . . . 305
20.9 Your First Demo: A Practical Roadmap . . . 306
The Build System . . . 309
21.2 Memory Map: 128K Bank Assignments . . . 310
21.4 The Gameplay Frame . . . 314

Reading Input . . . 315
21.5 Sprite Integration . . . 317
21.6 Collisions, Physics, and AI in Context . . . 319
Physics-Collision Loop . . . 319
Tile Collision . . . 319
21.8 Loading: Tape and DivMMC . . . 323
The .tap File and BASIC Loader . . . 323
21.9 Loading Screen, Menu, and High Scores . . . 326
Loading Screen . . . 326
21.11 Profiling with DeZog . . . 331
What is DeZog? . . . 331
The Profiling Workflow . . . 331
DeZog Configuration for Ironclaw . . . 333
21.12 The Data Pipeline in Detail . . . 333
Tilesets (PNG to Spectrum pixel format) . . . 334
Sprite Sheets (PNG to pre-shifted sprite data) . . . 334
Level Maps (Tiled JSON to binary tilemap) . . . 334
Music (Vortex Tracker II to PT3) . . . 334
Putting It Together . . . 335
21.13 Release Format: Building the .tap . . . 335
Summary . . . 337

Chapter 22: Porting — Agon Light 2 339
Same ISA, Different World . . . 339
The Architecture at a Glance . . . 340
The Practical Rule . . . 342
The MBASE Trap . . . 342
What Transfers Directly . . . 343
Game Logic and Entity System . . . 343
AABB Collision Detection . . . 344
Fixed-Point Arithmetic . . . 344
State Machine . . . 344
What Needs Rewriting . . . 345
Rendering: From Framebuffer to VDP Commands . . . 345
What Needs Rethinking . . . 347
Memory Architecture . . . 347
Loading . . . 348
Sound . . . 349
Input . . . 350
eZ80 at 18 MHz: What Still Matters, What Does Not . . . 350
What Still Matters: Inner Loop Efficiency . . . 350
What Becomes Irrelevant: Stack Tricks for Rendering . . . 351
The Comparison Table . . . 352
The Porting Process: Step by Step . . . 352
Step 1: Set Up the Agon Project . . . 353
Summary . . . 355

Chapter 23: AI-Assisted Z80 Development 356
23.1 The Historical Parallel: HiSoft C on ZX Spectrum . . . 356
23.2 The Claude Code Feedback Loop . . . 357
The Loop . . . 357
Where AI Helped Build MinZ . . . 362

Where AI Did Not Help Build MinZ . . . 363
The MinZ Verdict . . . 363
23.6 Honest Take: "Z80 They Still Don't Know" . . . 364
What the AI Gets Wrong, Specifically . . . 364
Where Introspec Is Right . . . 365
Where Introspec Is Not Quite Right . . . 366
23.7 The "Antique Toy" Demo: AI in Practice . . . 366
23.8 The Feedback Loop in Practice . . . 367
23.9 Building Your Own AI-Assisted Workflow . . . 367
Prompt Engineering for Z80 . . . 367
23.10 The Broader Picture . . . 368
Summary . . . 368
Try It Yourself . . . 369

Glossary 370
A. Timing & Performance . . . 370
B. Hardware — Sinclair & Clones . . . 372
C. Hardware — Soviet/Post-Soviet Ecosystem . . . 375
D. Hardware — Agon Light 2 . . . 376
E. Techniques . . . 377
F. Assembly Notation & Directives . . . 381
G. Demoscene & Culture . . . 383
Key People . . . 385
Key Demos & Productions . . . 386
Key Publications . . . 386
H. Algorithms & Compression . . . 386

Appendix A: Z80 Instruction Quick Reference 389
8-Bit Load Instructions . . . 389
16-Bit Load Instructions . . . 390
8-Bit Arithmetic and Logic . . . 390
16-Bit Arithmetic . . . 392
Rotate and Shift . . . 392
Bit Manipulation . . . 394
Jump, Call, Return . . . 394
I/O Instructions . . . 395
Block Instructions . . . 396
Exchange and Misc . . . 397
The Demoscene "Fast" Instructions . . . 399
Fastest Register-to-Register Move . . . 399
Fastest Way to Zero a Register . . . 399
Fastest Memory Read . . . 399
Fastest Memory Write . . . 399
Fastest 2-Byte Write . . . 399
Fastest 2-Byte Read . . . 399
Fastest Block Copy . . . 400
Fastest I/O . . . 400
Fastest Pointer Swap . . . 400
Fastest Conditional Loop . . . 400
Fastest Indirect Jump . . . 400
Undocumented Instructions . . . 400
IXH, IXL, IYH, IYL (Half-Index Registers) . . . 400

SLL r (Shift Left Logical) . . . 401
OUT (C),0 . . . 401
CB-Prefix Undocumented Bit Operations on (IX+d) . . . 401
Flag Effects Cheat Sheet . . . 402
Instructions That Set All Arithmetic Flags (S, Z, H, P/V, N, C) . . . 402
Instructions That Set Z and S (but NOT Carry) . . . 402
Instructions That Set ONLY Carry-Related Flags . . . 402
Instructions That Set NO Flags . . . 402
Practical Tricks . . . 402
Register Architecture . . . 403
Main Register Set . . . 403
Special Registers . . . 403
Shadow Registers . . . 403
Register Pairing for Instructions . . . 403
Common Instruction Sequences . . . 404
Pixel Address Calculation (Screen Address from Y,X) . . . 404
DOWN_HL: Move One Pixel Row Down . . . 404
8x8 Unsigned Multiply (Shift-and-Add) . . . 405
AY Register Write . . . 405
16-Bit Compare (HL vs DE) . . . 406
Stack-Based Screen Fill . . . 406
Fast Pixel-Row Iteration (Split Counters) . . . 406
Quick Cost Comparisons . . . 407
Instruction Encoding Size Reference . . . 408

Appendix B: Sine Table Generation and Trigonometric Tables 409
The Standard Format . . . 409
Comparison of Approaches . . . 410
Approach 1: Full 256-Byte Table . . . 410
Approach 2: Quarter-Wave Table . . . 411
Approach 3: Parabolic Approximation (Dark's Method) . . . 412
Approach 4: Second-Order Delta Encoding (The Deep Trick) . . . 412
The Key Insight . . . 412
Full Table via 2-Bit Second-Order Deltas . . . 413
Quarter-Wave via 2-Bit Second-Order Deltas . . . 413
Approach 5: Bhaskara I Approximation (7th Century) . . . 414
The Formula . . . 415
Accuracy . . . 415
Z80 Implementation . . . 415
Bhaskara I + Correction Bitmap (Exact) . . . 416
When to Use Bhaskara I . . . 416
Practical Recommendations . . . 416
The Decision Tree . . . 417
What Does Not Work . . . 417
Raider's Commandments . . . 417
Reference: The Full 256-Byte Table . . . 418

Appendix C: Compression Quick Reference 420
Compressor Comparison Table . . . 420
Decision Tree: Which Compressor? . . . 424
Compressibility of Common ZX Spectrum Data Types . . . 424
Pre-compression tricks . . . 427

Minimal RLE Decompressor . . . 427
ZX0 Standard Decompressor (Z80) . . . 428
Integration Patterns . . . 430
Pattern 1: Decompress to screen at startup . . . 430
Pattern 2: Decompress to buffer between effects . . . 430
Pattern 3: Stream decompression during playback . . . 430
Pattern 4: Bank-switched compressed data (128K) . . . 431
Build Pipeline: From Asset to Binary . . . 431
Quick Formulas . . . 432
See Also . . . 432

Appendix D: Development Environment Setup 434
1. The Assembler: sjasmplus . . . 434
Installing from Source . . . 434
Version Pinning . . . 435
Key Flags . . . 435
File Extension . . . 435
Hex Notation . . . 435
2. The Editor: VS Code . . . 436
Essential Extensions . . . 436
Build Task . . . 436
Recommended Settings . . . 437
3. Emulators . . . 437
ZEsarUX – Feature-Rich Debugging . . . 437
Fuse – Lightweight and Accurate . . . 438
Unreal Speccy – Windows, Pentagon-Focused . . . 438
For Agon Light 2 . . . 438
Which Emulator Should I Use? . . . 439
4. The Debugger: DeZog . . . 439
Installation . . . 439
Connecting to ZEsarUX . . . 439
Using the Internal Simulator . . . 440
Key DeZog Features . . . 441
5. Building the Book's Examples . . . 441
Clone the Repository . . . 441
Prerequisites . . . 441
Build Commands . . . 441
Running an Example . . . 442
6. Project Structure for Your Own Code . . . 442
Minimal Makefile . . . 443
Include Convention . . . 443
7. Alternative Tools . . . 444
Other Assemblers . . . 444
Other VS Code Extensions . . . 444
CSpect – Next-Focused Emulator . . . 445
SpectrumAnalyzer . . . 445
8. Troubleshooting . . . 445
"sjasmplus: command not found" . . . 445
Compilation errors in the book's examples . . . 445
DeZog cannot connect to ZEsarUX . . . 445
Emulator shows garbled screen . . . 446

Build output is empty or zero bytes . . . 446
Tool Reference . . . 446
See Also . . . 447

Appendix E: eZ80 Quick Reference 448
1. Architecture Overview . . . 448
2. Mode System . . . 449
The Two Modes . . . 449
Mode Suffixes . . . 449
Mode-Switching Calls and Jumps . . . 449
The Practical Rule . . . 450
The MBASE Trap . . . 450
3. New Instructions . . . 450
Arithmetic and Test . . . 450
Address Computation . . . 450
I/O and System . . . 451
4. MLT — The Game Changer . . . 451
What MLT Enables . . . 452
MLT Limitations . . . 452
5. Key Differences Summary . . . 452
6. Agon Light 2 Specifics . . . 453
Hardware . . . 453
Frame Budget . . . 453
MOS API . . . 454
VDP Command Protocol . . . 454
7. Porting Checklist . . . 455
See Also . . . 455

Appendix F: Z80 Variants — Extended Instruction Sets 457
1. The Z80 Family Tree . . . 457
2. Z80N — The Demoscener's Wish List . . . 458
Screen Navigation (the DOWN_HL problem) . . . 458
Sprite Rendering (the masked blit problem) . . . 459
Arithmetic (the multiply problem) . . . 459
Bit Manipulation . . . 460
Barrel Shifts (the multi-bit shift problem) . . . 460
Convenience Instructions . . . 461
The Big Picture . . . 461
3. eZ80 — The Enterprise Extension . . . 462
4. R800 — The MSX turboR Speedster . . . 463
5. Soviet Clones — Behind the Iron Curtain . . . 463
6. Comparison Table . . . 464
Effective Multiply Performance . . . 465
7. What This Means for the Book . . . 465
See Also . . . 465

Appendix G: AY-3-8910 / TurboSound / Triple AY Register Reference 467
I/O Ports on ZX Spectrum 128K . . . 467
The Write Sequence . . . 467
Reading a Register . . . 468
Bulk Register Write . . . 468
Complete Register Map . . . 468

Overview . . . 468
R0-R1: Channel A Tone Period (12-bit) . . . 470
R2-R3: Channel B Tone Period (12-bit) . . . 470
R4-R5: Channel C Tone Period (12-bit) . . . 470
R6: Noise Period (5-bit) . . . 470
R7: Mixer Control (The Most Important Register) . . . 471
R8-R10: Volume Registers (5-bit) . . . 472
R11-R12: Envelope Period (16-bit) . . . 472
R13: Envelope Shape (4-bit, Write-Only) . . . 473
Tone Period to Frequency Conversion . . . 475
Formula . . . 475
AY Clock by Platform . . . 475
Complete Note Frequency Table . . . 476
Compact Note Table for Z80 Code . . . 479
Table #5: Natural Tuning (Just Intonation) . . . 479
TurboSound: 2 x AY . . . 481
Chip Selection . . . 481
TurboSound Engine Architecture . . . 482
Stereo Configuration . . . 482
ZX Spectrum Next: Triple AY (3 x AY) . . . 483
Chip Selection on Next . . . 483
Per-Channel Stereo Panning . . . 483
Common Patterns in Z80 Code . . . 484
Silence the AY . . . 484
Play a Single Note on Channel A . . . 484
Trigger a Buzz-Bass Note . . . 484
Trigger Envelope (Restart) . . . 485
Digital Drum Hit (4-bit DAC) . . . 485
Tracker Format Notes . . . 486
ProTracker 3 (.pt3) . . . 486
Other Tracker Formats . . . 487
Vortex Tracker II . . . 488
AY-3-8910 vs YM2149: Differences That Matter . . . 488
Quick Reference Card . . . 488
Register Summary (Tear-Out) . . . 488

Appendix H: Storage APIs — TR-DOS and esxDOS 490
1. TR-DOS (Beta Disk 128) . . . 490
Hardware . . . 490
Disk Format . . . 490
WD1793 Port Map . . . 491
WD1793 Commands . . . 491
ROM API: Loading a File . . . 492
Direct Sector Access . . . 492
Disk Detection . . . 493
2. esxDOS (DivMMC / DivIDE) . . . 494
Hardware . . . 494
API Pattern . . . 494
Function Reference . . . 494
File Open Modes . . . 495
Seek Whence Values . . . 495

Code Example: Load a File 495
Code Example: Streaming Data from File 496
Detecting esxDOS 497
3. +3DOS (Amstrad +3) 498
4. Practical Patterns 498
Loading Screen from Disk (TR-DOS) 498
Loading Screen from SD (esxDOS) 499
Dual-Mode Loader 499
Streaming Compressed Data 500
5. File Format Reference 500
6. See Also 501

Appendix I: Bytebeat and AY-Beat – Generative Sound on Z80 502
1. Classic Bytebeat: The PCM Tradition 502
Famous Formulas 502
Why It Works 503
On the Spectrum: The Beeper Dead End 503
2. AY-Beat: Bytebeat Reimagined for a Tone Generator 504
Basic AY-Beat Architecture 504
What Changes from PCM Bytebeat 505
3. Drone: Envelope + Tone (E+T Mode) 505
The Drone Recipe 506
Byte Cost 507
4. Noise Percussion 507
Basic Kick Pattern 507
Percussion Character by Noise Period 508
Rhythmic Variety from Bit Masks 508
Using the Envelope for Drum Decay 509
5. Multi-Channel Harmony 509
Three Voices from One Formula 509
Why Rotation Creates Harmony 510
Volume Formulas for Multi-Channel 510
6. Formula Cookbook 511
Tone Period Formulas 511
Volume Formulas 512
How to Read the Table 513
7. Putting It Together: A Complete AY-Beat Engine 513
What This Produces 515
Customisation Points 515
8. Advanced: Combining Techniques 516
Architecture for a 256-Byte Intro 516
Register Budget 516
Size Breakdown 516
9. AY-as-DAC: Classic Bytebeat Through the Volume Register 517
Three Output Paths Compared 518
10. Music Theory for Algorithms 518
Scale Tables: Constraining Output to Pleasant Notes 518
Octave Derivation: Free Pitch Range 519
Arpeggio: Chord Tones in Sequence 520
Step Ornaments: Trills, Mordents, and Slides 520
Chord Progressions: Harmonic Movement 521

Total Data Budget for Rich Music . . . 521
11. L-System Grammars: Fractal Melodies . . . 522
The Concept . . . 522
Why L-Systems Work for Music . . . 522
Useful L-System Rules . . . 523
Z80 Implementation . . . 523
Melody as Motion, Not Absolute Notes . . . 525
Tribonacci: Three Symbols for Richer Patterns . . . 526
PRNG Melodies with Curated Seeds . . . 526
Combining L-Systems with Other Techniques . . . 527
Other Grammars for Music . . . 527
See Also . . . 527

What's New **529**
v12 (2026-02-24) . . . 529
v11 (2026-02-24) . . . 529
v10 (2026-02-24) . . . 530
v0.8 (2026-02-24) . . . 530
v0.7 (2026-02-24) . . . 531
v0.6 (2026-02-23) . . . 531

!!! UNDER CONSTRUCTION !!!

This book is a work in progress.
Content is incomplete, may contain errors,
and is subject to change without notice.

How this book is written:

1. With assistance from LLM (Claude Opus 4.6) based on open publications and data
2. By Alice directly — where personal expertise is “good enough” or no available sources or in new areas
3. Corrections and contributions on topic are welcome — PRs open

Factual errors are possible. If you spot one, please open an issue or PR.

Chapter 1: Thinking in Cycles

> "Coder effects are always about evolving a computation scheme." – Introspec (spke), Life on Mars

You have 71,680 clock ticks. That is your canvas, your budget, your entire world. Every instruction you write costs some of those ticks. Every frame, the counter resets and you get another 71,680 – no more, no less. Miss the deadline and the screen tears, the music stutters, the illusion breaks.

This chapter is about learning to see your code the way a Z80 demoscener does: not as text, not as algorithms, but as a *budget*.

T-States: The Currency of the Z80

The Z80 CPU does not execute instructions in uniform chunks. Every instruction takes a specific number of **T-states** (time states) – the fundamental clock cycles of the processor. At 3.5 MHz, one T-state lasts approximately 286 nanoseconds. That number is not important. What is important is that instructions have wildly different costs, and you need to know those costs by heart.

Here is a handful of instructions you will use constantly:

Instruction	What it does	T-states
`NOP`	Nothing	4
`LD A,B`	Copy B into A	4
`LD A,(HL)`	Load byte from memory address in HL	7
`LD (HL),A`	Store A to memory address in HL	7
`LD A,n`	Load an immediate byte into A	7
`INC HL`	Increment HL	6
`ADD A,B`	Add B to A	4
`PUSH HL`	Push HL onto the stack	11
`DJNZ label`	Decrement B, jump if not zero	13 (taken) / 8 (falls through)
`LDIR`	Block copy, per byte	21 (repeating) / 16 (last byte)
`OUT (n),A`	Write A to I/O port	11

Look at the range. A register-to-register `LD A,B` costs 4 T-states – the minimum for any instruction. A memory read `LD A,(HL)` costs 7, because the CPU needs extra machine cycles to put the address on the bus and wait for the RAM to respond. `LDIR`, the block copy instruction that every Spectrum coder reaches for instinctively, costs 21 T-states per byte it copies (except the last byte, which costs 16). That is over five times the cost of a NOP.

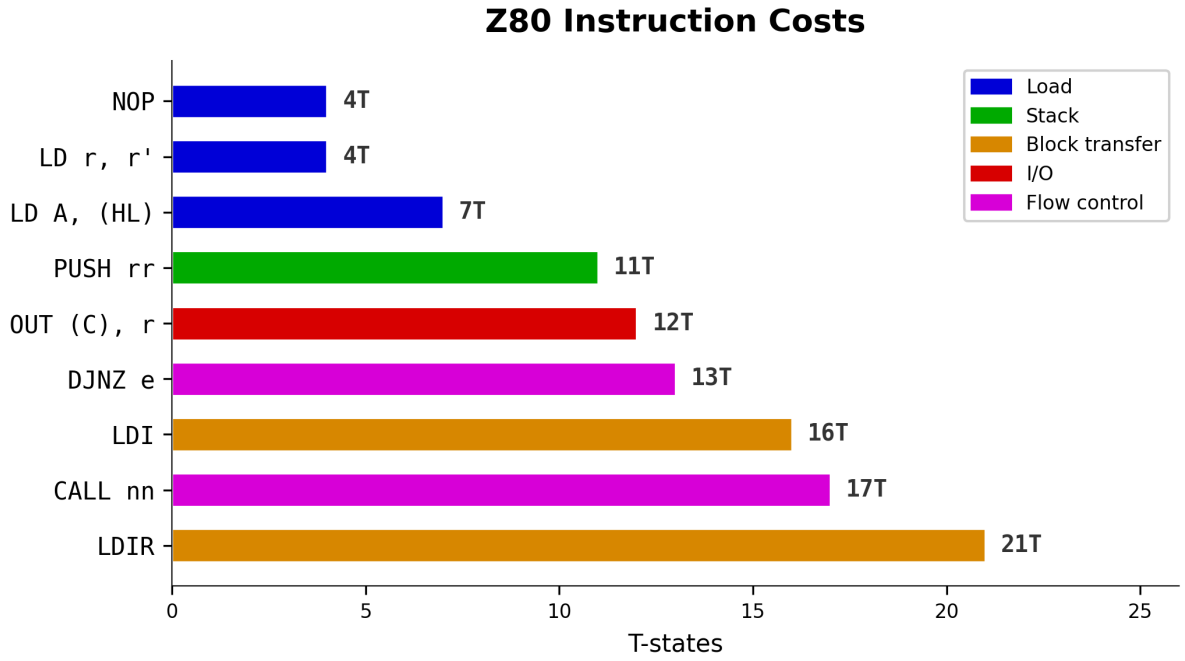


Figure 1: T-state costs for common Z80 instructions

Why does this matter? Because when you are filling a screen, or updating sprite data, or computing the next frame of a plasma effect, every instruction is eating into your budget. The difference between a 4-T-state instruction and a 7-T-state instruction, multiplied across ten thousand iterations in an inner loop, is the difference between an effect that runs at 50 frames per second and one that does not.

Machine Cycles and Memory Access

Each T-state is one tick of the CPU clock, but the Z80 does not talk to memory on every tick. Instructions are broken into **machine cycles** (M-cycles), each of which takes 3-6 T-states. The first machine cycle of every instruction is the **opcode fetch** (M1), which always takes 4 T-states: the CPU puts the program counter on the address bus, reads the opcode byte, and simultaneously refreshes DRAM. Further machine cycles read additional bytes (operands, memory data) or write results.

This is why `LD A,B` takes exactly 4 T-states – it is a single-byte instruction that completes entirely within the opcode fetch. But `LD A,(HL)` takes 7 T-states: 4 for the opcode fetch, then 3 more for the memory read cycle where the CPU puts HL on the address bus and reads the byte at that address.

You do not need to memorize the internal machine cycle breakdown of every instruction. But understanding the pattern – opcode fetch + operand reads + memory accesses = total cost – helps you develop intuition for *why* instructions cost what they cost. A `PUSH HL` at 11 T-states makes sense when you realise the CPU

must do an opcode fetch (5T for this one, since it also decrements SP), then two separate memory write cycles (3T each) to store the high and low bytes of HL onto the stack.

The Frame: Your Canvas

The ZX Spectrum generates a PAL video signal at approximately 50 frames per second. Every frame, the ULA chip reads video memory and paints the screen, line by line. At the end of each frame, the ULA fires a maskable interrupt. The CPU executes a `HALT` instruction to wait for that interrupt, does its work, and then `HALT`s again to wait for the next frame. This is the heartbeat of every Spectrum program.

The number of T-states between one interrupt and the next – the **frame budget** – depends on the machine:

Machine	T-states per frame	Scanlines	Hz
ZX Spectrum 48K	69,888	312	50.08
ZX Spectrum 128K	70,908	311	50.02
Pentagon 128	71,680	320	48.83

Those are *total* T-states between interrupts. The practical budget is less – subtract the interrupt handler cost (a PT3 music player typically consumes 3,000–5,000 T-states per frame), HALT overhead, and on non-Pentagon machines, contention penalties. On a Pentagon with a music player, expect roughly 66,000–68,000 T-states for your main loop. Chapter 15 has the detailed tact-maps.

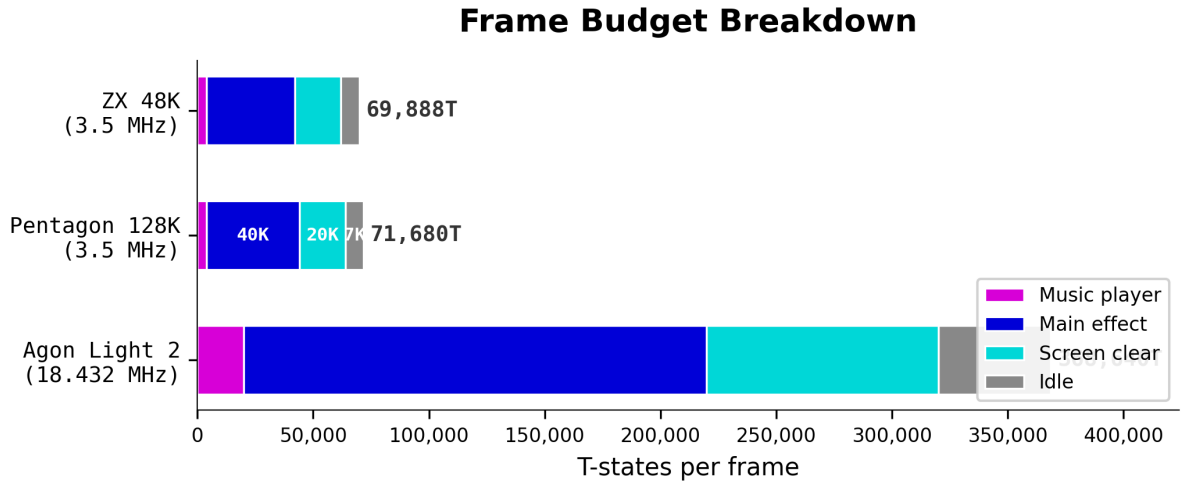


Figure 2: Frame budget breakdown across ZX Spectrum models

If your main loop – input handling, game logic, sound update, screen rendering – takes more T-states than one frame, you drop frames. Things slow down. The border stripe trick we will build later in this chapter will make that painfully visible.

To put these numbers in perspective: a single `LDIR` copying 6,912 bytes (one full screen of pixel data) costs approximately 6,912 x 21 = 145,152 T-states. That is

more than two entire frames on a 48K Spectrum. You cannot even copy the screen once per frame with the simplest possible method. This is the kind of constraint that forces creativity.

Pentagon vs. Wait Machines

You will notice that the frame budgets above differ between machines. The difference is not just in the numbers – it reflects a fundamental architectural split that shaped the ZX Spectrum demoscene.

The Original Sinclair Machines

On the original 48K and 128K Spectrums, the screen memory lives at addresses `$4000-$5AFF` (pixel data) and `$5800-$5B00` (colour attributes). This memory region – the entire `$4000-$7FFF` range, in fact – is **contended memory**. The ULA (Uncommitted Logic Array), which generates the video signal, needs to read this memory to paint the screen. The CPU and the ULA share the same memory bus, and when both want to read at the same time, the ULA wins. The CPU is forced to wait.

During the 192 active display lines, every CPU access to the `$4000-$7FFF` range is potentially delayed. The delay follows a repeating 8-T-state pattern: 6, 5, 4, 3, 2, 1, 0, 0 extra wait states, cycling across each scanline. An instruction that should take 7 T-states might take 13 if it lands on the worst phase of the contention cycle.

This makes cycle-counting on original Spectrums a nightmare. Your carefully calculated inner loop runs at a different speed depending on where in the frame it executes, and whether the code or data it touches happens to be in the contended range. Introspec documented this in his "GO WEST" articles on Hype (2015): during screen rendering, each byte access to contended memory costs an average of 2.625 extra T-states. For stack operations writing to screen memory, expect roughly 1.3 extra T-states per byte.

The Pentagon: Clean Timing

The Pentagon 128, the most popular Soviet ZX Spectrum clone, took a different approach. Its designers gave the ULA its own memory access window that does not conflict with the CPU. **There is no contended memory on the Pentagon.** Every instruction takes exactly the number of T-states listed in the datasheet, regardless of where the code lives or what memory it accesses.

This is why the Pentagon has a different frame length – 71,680 T-states, 320 scanlines. The ULA timing is slightly different because there is no need to interleave CPU and ULA access. But the payoff is enormous: you can count cycles with absolute confidence. When your inner loop says it costs 36 T-states per iteration, it costs 36 T-states per iteration, every single time, everywhere in the frame.

This clean timing is why the Pentagon became the standard platform for the ZX Spectrum demoscene, particularly in the Former Soviet Union where these clones were ubiquitous. When you watch demos from groups like X-Trade, 4th Dimension, or Life on Mars, they are overwhelmingly targeting Pentagon timing. When

Introspec wrote his legendary technical teardown of Illusion by X-Trade, the cycle counts he quoted assumed Pentagon.

For learning, the Pentagon model is ideal: you can focus on understanding what instructions cost without worrying about contention effects. All the T-state tables in this book assume Pentagon timing unless stated otherwise. When we need to discuss the differences (and we will, in Chapter 15), we will be explicit.

The practical rule: place your time-critical code in uncontended memory (`$8000-$FFFF` on a 48K), and your cycle counts will be correct on both Pentagons and original Spectrums.

Thinking in Budgets

Now that you know the frame size, you can start doing the arithmetic that defines Z80 demoscene thinking.

Say you want to fill the entire screen with a calculated colour every frame – a simple plasma effect, updating only the 768 bytes of attribute memory at `$5800`. At 50 fps, you need to compute and write 768 colour values every 71,680 T-states.

If your inner loop per attribute byte looks like this:

```
z80 id:ch01_thinking_in_budgets     ld   a,c        ; 4T   column index     add
a,b        ; 4T   add row index (diagonal pattern)     add  a,d        ; 4T
add frame counter (animation)     and  7          ; 7T   clamp to 0-7     ld
(hl),a     ; 7T   write attribute     inc  hl         ; 6T   next attribute
address                     ; --- 32T per byte
```

That is 32 T-states per byte. For 768 bytes: 32 x 768 = 24,576 T-states. Add loop overhead (maintaining counters for rows and columns, the `DJNZ` for the inner loop), and you might land around 28,000-30,000 T-states. That leaves over 40,000 T-states for everything else – music playback, input handling, whatever you need.

But what if you wanted to update every *pixel* byte, all 6,144 of them? At 32 T-states per byte, that is 196,608 T-states – nearly three frames. Suddenly you are looking at a 17 fps update rate instead of 50 fps. You either need a faster inner loop, a smaller update region, or a different approach entirely.

This is how Z80 programmers think. Every design decision starts with arithmetic: how many bytes, how many T-states per byte, how many T-states in the frame budget, does it fit? When it does not fit, you do not reach for a faster machine – you reach for a cleverer algorithm.

> **Agon Light 2 Sidebar**
>
> The Agon Light 2 runs a Zilog eZ80 at 18.432 MHz. The eZ80 executes the same Z80 instruction set (it is a direct architectural descendant), but most instructions run in fewer clock cycles – many single-byte instructions complete in just 1 cycle instead of 4. At 18.432 MHz with a 50 Hz frame rate, you get approximately **368,640 T-states per frame**.

That is just over 5 times the Pentagon's budget. The same Z80 assembly language, the same registers, the same instruction mnemonics – but with five times the room to breathe. An inner loop that consumes 70% of a Pentagon frame might use only 14% of an Agon frame.

This does not make the Agon "easy." It has its own constraints: no ULA-style video memory (the display is managed by an ESP32 coprocessor running the VDP), flat 24-bit addressing instead of banked memory, and a completely different I/O model. But if you have ever wished you could just have a *little more room* in your frame budget to try something ambitious, the Agon is where the same Z80 thinking scales up.

Throughout this book, we will note where the Agon's larger budget changes the calculus. For now, just remember the number: **~368,000 T-states**. Same ISA, five times the canvas.

Practical: Setting Up Your Development Environment

Before we write our first timing harness, you need a working toolchain. The setup described here follows sq's guide from Hype (2019), which has become the community standard.

What You Need

1. **VS Code** – your editor and integrated environment.
2. **Z80 Macro Assembler extension** by mborik (`mborik.z80-macroasm`) – syntax highlighting, code completion, symbol resolution for Z80 assembly. Install from the VS Code marketplace.
3. **Z80 Assembly Meter** by Nestor Sancho – displays the byte count and cycle count of the currently selected instruction(s) in the status bar. This is invaluable. Select a block of code and see its total T-state cost instantly.
4. **sjasmplus** – the assembler itself. Cross-platform, open source, supports macros, Lua scripting, multiple output formats. Download from https://github.com/z00m128/sjasmplus and place the binary somewhere in your PATH.
5. **Unreal Speccy** (Windows) or **Fuse** (cross-platform) – the emulator. Unreal Speccy is preferred for demo development because it emulates Pentagon timing accurately and has a built-in debugger.

Project Structure

Create a directory for your Chapter 1 experiments. A minimal project looks like this:

```
ch01/
  main.a80          -- your source file
  build.bat         -- (Windows) sjasmplus main.a80
  Makefile           -- (macOS/Linux) make target
```

Build Configuration

In VS Code, set up a build task (`.vscode/tasks.json`) so you can compile with Ctrl+Shift+B:

```
{
  "version": "2.0.0",
  "tasks": [
    {
      "label": "Assemble Z80",
      "type": "shell",
      "command": "sjasmplus",
      "args": [
        "--fullpath",
        "--nologo",
        "--msg=war",
        "${file}"
      ],
      "group": {
        "kind": "build",
        "isDefault": true
      },
      "problemMatcher": {
        "owner": "z80",
        "fileLocation": "absolute",
        "pattern": {
          "regexp": "^(.*)\\((\\d+)\\):\\s+(error|warning):\\s+(.*)$",
          "file": 1,
          "line": 2,
          "severity": 3,
          "message": 4
        }
      }
    }
  ]
}
```

Press Ctrl+Shift+B. If sjasmplus is in your PATH and there are no errors, you will get a `.sna` or `.tap` file (depending on your source directives) that you can open directly in your emulator.

For integration with Unreal Speccy, Alex_Rider's 2024 extension adds an F5-to-launch binding – the emulator opens your compiled snapshot automatically. If you are on macOS or Linux and using Fuse, a simple Makefile rule does the same:

```
run: main.sna
    fuse --machine pentagon main.sna
```

Practical: The Timing Harness

This is the most important debugging tool you will build in this entire book. It is dead simple, it requires no special hardware, and you will use it constantly.

The idea: change the border colour to red immediately before the code you want to measure, and change it back to black immediately after. The Spectrum's border is drawn by the ULA in real time, synchronised with the electron beam. A wider red stripe means more T-states spent in your code.

Here is the complete harness:

"'z80 id:ch01_practical_the_timing_harness ORG $8000

start: ; Wait for the frame interrupt halt

```
; --- Border RED: code under test begins ---
ld   a, 2          ; 7T  red = colour 2
out  ($FE), a      ; 11T write to border port

; ===== CODE UNDER TEST =====
; Replace this block with whatever you want to measure.
; Example: 256 iterations of a NOP loop.

ld   b, 0          ; 7T  B=0 wraps to 256 iterations
```

.loop: nop ; 4T nop ; 4T nop ; 4T nop ; 4T – 16T per iteration body djnz .loop ; 13T taken, 8T on final iteration ; Total: 256 * (16+13) - 5 = 7,419 T-states

```
; ===== END CODE UNDER TEST =====

; --- Border BLACK: idle ---
xor  a             ; 4T  A=0 (black), shorter than LD A,0
out  ($FE), a      ; 11T

; Loop forever
jr   start

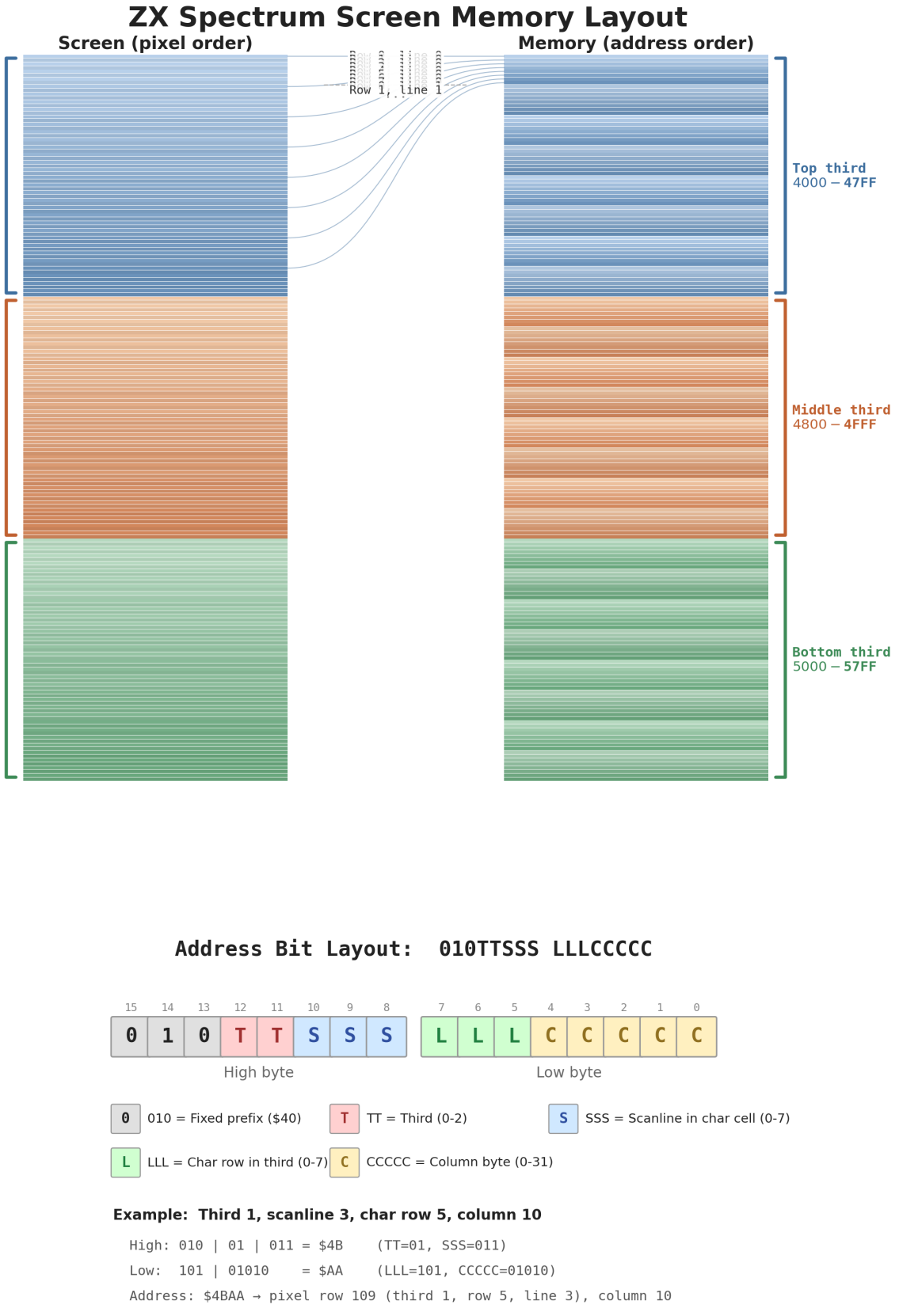

Load this into your emulator. You will see a red stripe across the border. The height of that
states your test code consumed.

### Reading the Stripe

Each scanline takes 224 T-states (on Pentagon). So if your red stripe is N scanlines tall, you
states. The example above uses about 7,419 T-states, which is roughly 33 scanlines -- you shou
sixth of the way down the border.

Now try replacing the NOP loop with something heavier. Replace the four NOPs with:

```z80 id:ch01_reading_the_stripe
.loop:
    ld   a,(hl)        ; 7T
    add  a,(hl)        ; 7T
    ld   (de),a        ; 7T
    inc  hl            ; 6T   -- 27T per iteration body
    djnz .loop         ; 13T taken
    ; Total: 256 * (27+13) - 5 = 10,235 T-states
```

The red stripe grows noticeably. That visual difference – you can see it without a
```

debugger, without a profiler, without any tooling at all – is 2,816 T-states. About 12 scanlines.

This is how Spectrum demo coders have profiled their effects since the 1980s. The border is your oscilloscope.

Variations

You can use different colours to mark different phases of your code:

```
z80 id:ch01_variations     ld   a, 2          ; red     out  ($FE), a     call
render_sprites     ld   a, 1          ; blue     out  ($FE), a     call update_music
ld   a, 4          ; green     out  ($FE), a     call game_logic     xor  a
; black     out  ($FE), a
```

Now the border shows a red band (sprite rendering), then blue (music), then green (game logic), then black (idle time). You can see at a glance which subsystem is eating your frame budget.

A note on `xor a` versus `ld a, 0`: both set A to zero. `XOR A` takes 4 T-states and 1 byte. `LD A, 0` takes 7 T-states and 2 bytes. In a timing harness the difference is negligible, but it is worth noticing – this kind of micro-awareness is what Z80 coding is made of.

What Fits in a Frame?

Let us use our budget arithmetic to answer some practical questions.

How many sprites can you draw per frame? A 16x16 masked sprite using the OR+AND method takes roughly 16 scanlines x (read mask + read sprite + read screen + combine + write screen) per byte. A reasonable estimate is about 1,200 T-states per sprite. On a Pentagon, that is 71,680 / 1,200 = ~59 sprites, if sprite rendering were the *only* thing you did. In practice, with music, game logic, and everything else, 8-12 full sprites per frame is typical.

How many bytes can LDIR copy per frame? At 21 T-states per byte: 71,680 / 21 = 3,413 bytes. Not even half the screen.

How many multiply operations? A fast square-table 8x8 multiply takes about 54 T-states. 71,680 / 54 = 1,327 multiplications per frame. A single-point 3D rotation needs 9 multiplies. So you could rotate about 147 points per frame *if you did nothing else*. Practical limit with a full demo engine: 30-50 points.

Every design question reduces to this arithmetic. Can I do it? How many can I do? What do I have to give up to make room?

Historical Note: Dark's Advice

In 1997, a programmer named Dark from the group X-Trade published a series of articles in *Spectrum Expert* #01, a Russian electronic magazine for ZX Spectrum developers. These articles covered multiplication, division, sine/cosine generation,

and line-drawing algorithms in Z80 assembly – the fundamental building blocks that power every demo effect.

Dark opened with this advice:

> "Read a maths textbook – derivatives, integrals. Knowing them, you can create a table of practically any function in assembly."

This was not empty theory. Dark was not just a writer – he was a coder. X-Trade's demo *Illusion*, released at ENLiGHT'96, featured a textured spinning sphere, a rotozoomer, a 3D engine, and a bouncing dot scroller, all running on a 3.5 MHz Z80. The algorithms Dark described in his magazine articles were the same algorithms powering his demo's effects.

Twenty years later, Introspec (spke) published a detailed technical teardown of Illusion on Hype, analysing the inner loops instruction by instruction, counting every T-state. The magazine articles from 1997 and the reverse engineering from 2017 tell the same story from both sides: the author explaining his building blocks, and a peer measuring the finished machine. We will follow this thread throughout the book.

Dark's point stands: the maths is not optional. You do not need a degree in mathematics, but you need to understand how to turn a mathematical function into a table, how to approximate expensive operations with cheap ones, and how to think about error versus speed. Chapter 4 will walk through Dark's algorithms in detail. For now, remember his advice. It is the starting point of everything.

The Computation Scheme

Introspec, writing about what makes a good demo effect, distilled the philosophy into a single sentence:

> "Coder effects are always about evolving a computation scheme."

This is the deepest insight in this chapter. A demo effect is not a picture; it is a *process*. Each frame, the computation scheme produces the next state from the previous one. The art is in choosing a scheme that produces visually compelling evolution while fitting within the frame budget.

A plasma is a computation scheme: sum sine waves at each grid position, offset by time. A tunnel is a computation scheme: look up angle and distance from pre-computed tables, offset by time. A spinning 3D object is a computation scheme: multiply vertex coordinates by a rotation matrix that changes each frame. The particular scheme determines the visual result, the T-state cost, and the memory requirements – all at once, all interlocked.

When you sit down to write an effect, you are not asking "how do I draw this picture." You are asking "what computation, evolved frame by frame, produces this visual?" That shift in thinking – from image to process, from output to scheme – is the Z80 programmer's worldview.

And the first constraint on any scheme is the budget. 71,680 T-states. Can you evolve your computation within that budget? If not, can you find a cheaper scheme that produces a similar visual? Can you precompute part of the scheme into tables?

Can you spread the computation across multiple frames? Can you exploit symmetry to compute half the screen and mirror the other half?

These are the questions that drive every chapter in this book. They start here, with counting T-states.

Summary

- Every Z80 instruction has a specific T-state cost. Learn the common ones by heart: `NOP` = 4, `LD A,B` = 4, `LD A,(HL)` = 7, `PUSH HL` = 11, `LDIR` = 21/16, `OUT (n),A` = 11.
- The **frame budget** is your hard constraint: 69,888 T-states (48K), 70,908 (128K), or 71,680 (Pentagon). At 50 fps, everything must fit.
- **Pentagon has no contended memory**, making cycle counts reliable and predictable. This is why it became the demoscene standard.
- The **Agon Light 2** (eZ80 @ 18.432 MHz) gives ~368,000 T-states per frame – same instruction set, five times the room.
- The **border colour timing harness** is your oscilloscope: red before, black after, read the stripe width.
- Z80 programming is **budget arithmetic**: bytes x T-states per byte vs. frame budget. Every design decision starts here.
- Effects are **computation schemes evolved over time**. The art is finding a scheme that fits the budget and looks good.

Try It Yourself

1. Build the timing harness from this chapter. Replace the NOP loop with a `LDIR` that copies 256 bytes and compare the stripe width to the NOP version. Calculate the expected T-state difference and verify it visually.
2. Write a loop that fills all 768 bytes of attribute memory (`$5800-$5AFF`) with a single colour value. Measure it with the harness. Now try filling it using `LDIR` instead of a byte-by-byte loop. Which is faster? By how many scanlines?
3. Open the Z80 Assembly Meter in VS Code. Select different code blocks and watch the T-state counter in the status bar. Get used to checking costs as you write.
4. Set up the multi-colour border profiler (red / blue / green / black) with three dummy loops of different lengths. Adjust the loop counts until you can visually distinguish all three bands. This is your calibration exercise for reading border timing.

Next: Chapter 2 – The Screen as a Puzzle. We will dive into the Spectrum's notoriously scrambled video memory layout and learn why `INC H` moves you one pixel down.

Chapter 2: The Screen as a Puzzle

> "Why do the rows go in that order?" – Every ZX Spectrum programmer, at some point

Open any emulator, type `PEEK 16384` and you are reading the first byte of the Spectrum's screen. But which byte is it? Not the top-left of the screen in any simple sense. The pixel at coordinate (0,0) is there, yes – but the pixel at (0,1), the very next row down, lives 256 bytes away. The pixel at (0,8), the top row of the second character cell, lives only 32 bytes from the start. And the pixel at (0,64) – the first row of the screen's middle third – lives exactly 2,048 bytes from the start, at `$4800`.

This is the Spectrum's most famous puzzle. The screen memory layout is not sequential, not intuitive, and not an accident. It is a consequence of hardware design choices made in 1982, and it shapes every piece of code that touches the display. Understanding this layout – and learning the tricks that make it fast to navigate – is fundamental to everything that follows in this book.

The Memory Map: 6,912 Bytes of Screen

The Spectrum's display occupies a fixed region of memory:

```
$4000 - $57FF    Pixel data      6,144 bytes   (256 x 192 pixels, 1 bit per
↪  pixel)
$5800 - $5AFF    Attributes        768 bytes   (32 x 24 colour cells)
```

The pixel area holds the bitmap: 256 pixels across, packed 8 per byte, giving 32 bytes per row. With 192 rows, that is 32 x 192 = 6,144 bytes. Each byte represents 8 horizontal pixels, with bit 7 as the leftmost pixel and bit 0 as the rightmost.

The attribute area holds the colour information: one byte per 8x8 character cell. There are 32 columns and 24 rows, giving 32 x 24 = 768 bytes.

Together: 6,144 + 768 = 6,912 bytes. That is the entire display.

The pixel data and attribute data serve different purposes but are tightly coupled. Each pixel byte controls 8 dots on screen; the attribute byte for the corresponding 8x8 cell controls what colour those dots appear in. Change the pixel and you change the shape. Change the attribute and you change the colour. But you can only change colour for an entire 8x8 block – not per pixel. This is the "attribute clash" that defines the Spectrum's visual character, and we will return to it shortly.



Figure 3: ZX Spectrum screen memory layout with thirds, character cells, and attribute area

First, the puzzle: why are the pixel rows scrambled?

The Interleave: Where the Rows Live

If the Spectrum stored its pixel rows sequentially, row 0 would be at `$4000`, row 1 at `$4020`, row 2 at `$4040`, and so on. Each row is 32 bytes, so row N would simply be at `$4000 + N * 32`. Simple, fast, sensible.

That is not what happens.

The screen is divided into three **thirds**, each 64 pixel rows tall. Within each third, the rows are interleaved by character cell row. Here is where the first 16 rows actually live:

```
Row  0:  $4000     Third 0, char row 0, scan line 0
Row  1:  $4100     Third 0, char row 0, scan line 1
Row  2:  $4200     Third 0, char row 0, scan line 2
Row  3:  $4300     Third 0, char row 0, scan line 3
Row  4:  $4400     Third 0, char row 0, scan line 4
Row  5:  $4500     Third 0, char row 0, scan line 5
Row  6:  $4600     Third 0, char row 0, scan line 6
Row  7:  $4700     Third 0, char row 0, scan line 7
Row  8:  $4020     Third 0, char row 1, scan line 0
Row  9:  $4120     Third 0, char row 1, scan line 1
Row 10:  $4220     Third 0, char row 1, scan line 2
Row 11:  $4320     Third 0, char row 1, scan line 3
Row 12:  $4420     Third 0, char row 1, scan line 4
Row 13:  $4520     Third 0, char row 1, scan line 5
Row 14:  $4620     Third 0, char row 1, scan line 6
Row 15:  $4720     Third 0, char row 1, scan line 7
```

Look at the pattern. The first 8 rows are the 8 scanlines of character row 0 – but they are 256 bytes apart, not 32. Within those 8 rows, the high byte of the address increments by 1 each time: `$40`, `$41`, `$42`, … `$47`. Then row 8 jumps to `$4020` – back to a high byte of `$40`, but with the low byte advanced by 32.

Here is the complete picture for the top third of the screen:

```
Char row 0:   scan lines at $4000, $4100, $4200, $4300, $4400, $4500, $4600,
↪  $4700
Char row 1:   scan lines at $4020, $4120, $4220, $4320, $4420, $4520, $4620,
↪  $4720
Char row 2:   scan lines at $4040, $4140, $4240, $4340, $4440, $4540, $4640,
↪  $4740
Char row 3:   scan lines at $4060, $4160, $4260, $4360, $4460, $4560, $4660,
↪  $4760
Char row 4:   scan lines at $4080, $4180, $4280, $4380, $4480, $4580, $4680,
↪  $4780
Char row 5:   scan lines at $40A0, $41A0, $42A0, $43A0, $44A0, $45A0, $46A0,
↪  $47A0
Char row 6:   scan lines at $40C0, $41C0, $42C0, $43C0, $44C0, $45C0, $46C0,
↪  $47C0
```

```
Char row 7:   scan lines at $40E0, $41E0, $42E0, $43E0, $44E0, $45E0, $46E0,
↪  $47E0
```

The middle third starts at `$4800` and follows the same pattern. The bottom third starts at `$5000`.

Why?

The reason is the ULA – the Uncommitted Logic Array that generates the video signal. The ULA reads one byte of pixel data and one byte of attribute data for every 8-pixel character cell it draws. It needs both bytes at specific moments as it rasters across the screen.

The interleaved layout meant that the ULA's address counter logic could be built with fewer gates. As the ULA scans left to right across a character row, it increments the low 5 bits of the address (the column). When it reaches the right edge, it increments the high byte to move to the next scanline within the same character row. When it finishes all 8 scanlines, it wraps the high byte and advances the low-byte row bits.

This is elegant from a hardware perspective. The ULA's address generation is a simple combination of counters – no multiplication, no complex address arithmetic. The PCB routing was simpler, the gate count was lower, and the chip was cheaper to manufacture.

The programmer pays the price.

The Bit Layout: Decoding (x, y) into an Address

To understand the interleave precisely, look at how the Y coordinate maps into the 16-bit screen address. Consider a pixel at column `x` (0–255) and row `y` (0–191). The byte containing that pixel is at:

```
High byte:  0 1 0 T T S S S
Low byte:   L L L C C C C C
```

Where: - `TT` = which third of the screen (0, 1, or 2). Bits 7–6 of y. - `SSS` = scanline within the character cell (0–7). Bits 2–0 of y. - `LLL` = character row within the third (0–7). Bits 5–3 of y. - `CCCCC` = column in bytes (0–31). This is x / 8, or equivalently bits 7–3 of x.

The crucial thing: the bits of y are not in order. Bits 7-6 go to one place, bits 5-3 go to another, and bits 2-0 go to yet another. The y coordinate is sliced up and distributed across the address.

Let us visualise this with a concrete example. Pixel (80, 100):

```
x = 80:     column byte = 80 / 8 = 10      CCCCC = 01010
y = 100:    binary = 01100100
            TT  = 01       (third 1, the middle third)
            LLL = 100      (char row 4 within the third)
            SSS = 100      (scan line 4 within the char cell)

High byte:  0  1  0  0  1  1  0  0  = $4C
```

```
Low byte:   1  0  0  0  1  0  1  0  = $8A

Address: $4C8A
```

The bit within that byte is determined by the low 3 bits of x. Bit 7 is the leftmost pixel, so pixel position (x AND 7) maps to bit 7 - (x AND 7).

The address calculation in Z80

Converting (x, y) to a screen address is something you need to do fast and often. Here is a standard routine:

“‘z80 id:ch02_the_address_calculation_in ; Input: B = y (0-191), C = x (0-255) ; Output: HL = screen address, A = bit mask ; pixel_addr: ld a, b ; 4T A = y and $07 ; 7T A = SSS (scan line within char) or $40 ; 7T A = 010 00 SSS (add screen base) ld h, a ; 4T H = high byte (partial)

```
ld   a, b          ; 4T   A = y again
rra                ; 4T   \
rra                ; 4T    | shift bits 5-3 of y
rra                ; 4T   /  down to bits 2-0
and  $E0           ; 7T   mask to get LLL 00000
ld   l, a          ; 4T   L = LLL 00000 (partial)

ld   a, b          ; 4T   A = y again
and  $C0           ; 7T   A = TT 000000
rra                ; 4T   \
rra                ; 4T    | shift bits 7-6 of y
rra                ; 4T   /  to bits 4-3
or   h             ; 4T   combine with SSS
ld   h, a          ; 4T   H = 010 TT SSS (complete)

ld   a, c          ; 4T   A = x
rra                ; 4T   \
rra                ; 4T    | x / 8
rra                ; 4T   /
and  $1F           ; 7T   mask to CCCCC
or   l             ; 4T   combine with LLL 00000
ld   l, a          ; 4T   L = LLL CCCCC (complete)
                   ; --- Total: ~91 T-states


91 T-states is not cheap. In a tight inner loop processing thousands of pixels, you would not

---

## DOWN_HL: Moving One Pixel Row Down

You have a pointer in HL to some byte on the screen. You want to move it one pixel row down --

On a linear framebuffer, you add 32 (the number of bytes per row). One `ADD HL, DE` with DE =
states, done.
```

```
On the Spectrum, it is a puzzle within the puzzle. Moving one pixel row down means:

1. **Within a character cell** (scanlines 0--6 to 1--7): increment H. The scanline bits are in

2. **Crossing a character cell boundary** (scanline 7 to scanline 0 of the next row): reset th

3. **Crossing a third boundary** (bottom of char row 7 in one third to top of char row 0 in th

The classic routine handles all three cases:

```z80 id:ch02_downhl_moving_one_pixel_row
; DOWN_HL: move HL one pixel row down on the Spectrum screen
; Input:  HL = current screen address
; Output: HL = screen address one row below
;
down_hl:
    inc  h              ; 4T   try moving one scan line down
    ld   a, h           ; 4T
    and  7              ; 7T   did we cross a character boundary?
    ret  nz             ; 11/5T  no: done

    ; Crossed a character cell boundary.
    ; Reset scan line to 0, advance character row.
    ld   a, l           ; 4T
    add  a, 32          ; 7T   next character row (L += 32)
    ld   l, a           ; 4T
    ret  c              ; 11/5T  if carry, we crossed into next third

    ; No carry from L, but we need to undo the H increment
    ; that moved us into the wrong third.
    ld   a, h           ; 4T
    sub  8              ; 7T   back up one third in H
    ld   h, a           ; 4T
    ret                 ; 10T
```

This routine takes different amounts of time depending on which case it hits:

Case	Frequency	T-states
Within a character cell	7 out of 8 rows	4 + 4 + 7 + 11 = **26**
Character boundary, same third	7 out of 64 rows	4 + 4 + 7 + 5 + 4 + 7 + 4 + 5 + 4 + 7 + 4 + 10 = **65**
Third boundary	2 out of 192 rows	4 + 4 + 7 + 5 + 4 + 7 + 4 + 11 = **46**

The common case – staying within a character cell – is fast: 26 T-states (a conditional RET that fires costs 11T, not 5T). The uncommon case (crossing a character row boundary within the same third) is 65 T-states. Averaged over all 192 rows, the cost works out to about **30.5 T-states per call**.
```

That average hides a problem. If you are iterating down the full screen and calling DOWN_HL on every row, those occasional 65-T-state calls spike your per-line timing unpredictably. For a demo effect that needs consistent timing per scanline, this jitter is unacceptable.

Introspec's Optimisation

In December 2020, Introspec (spke) published a detailed analysis on Hype titled "Once more about DOWN_HL" (Eshchyo raz pro DOWN_HL). The article examined the problem of iterating down the full screen efficiently – not just the cost of one call, but the total cost of moving HL through all 192 rows.

The naive approach – calling the classic DOWN_HL routine 191 times – costs **5,825 T-states** for a full screen traversal. Introspec's goal was to find the fastest way to iterate through all 192 rows, visiting every screen address in top-to-bottom order.

His key insight was to use **split counters**. Instead of testing the address bits after every increment to detect boundary crossings, he structured the loop to match the screen's three-level hierarchy directly:

```
text id:ch02_introspec_s_optimisation For each third (3 iterations):     For
each character row within the third (8 iterations):         For each scan line
within the character cell (8 iterations):             process this row             INC
H                   ; next scan line         undo 8 INC H's, ADD 32 to L   ; next
character row     undo 8 ADD 32's, advance to next third
```

The innermost operation is just `INC H` – 4 T-states. No testing, no branching. The character-row and third transitions happen at fixed, predictable points in the loop, so there is no conditional logic in the inner loop at all.

The result: **2,343 T-states** for a full screen traversal. That is a 60% improvement over the classic approach, and the per-line cost is absolutely predictable – no jitter.

There was also an elegant variation attributed to RST7, using a dual-counter approach where the outer loop maintains a pair of counters that naturally track the character-row and third boundaries. The inner loop body reduces to a single `INC H`, and the boundary handling is folded into the counter manipulation at the outer loop level.

The practical lesson: when you need to iterate through the Spectrum's screen in order, do not call a general-purpose DOWN_HL routine 191 times. Restructure your loop to match the screen's natural hierarchy, and the branching disappears.

Here is a simplified version of the split-counter approach:

"'z80 id:ch02_introspec_s_optimisation_2 ; Iterate all 192 screen rows using split counters ; HL = $4000 at entry (top-left of screen) ; iterate_screen: ld hl, $4000 ; 10T start of screen ld c, 3 ; 7T 3 thirds

.third_loop: ld b, 8 ; 7T 8 character rows per third

.row_loop: push hl ; 11T save start of this char row

```
; --- Process 8 scan lines within this character cell ---
REPT 7
    ; ... your per-row code here, using HL ...
    inc  h              ; 4T   next scan line
```

```
ENDR
; ... process the 8th (last) scan line ...

pop  hl                 ; 10T  restore char row start
ld   a, l               ; 4T
add  a, 32              ; 7T   next character row
ld   l, a               ; 4T

djnz .row_loop          ; 13T/8T

; Advance to next third
ld   a, h               ; 4T
add  a, 8               ; 7T   next third ($0800 higher)
ld   h, a               ; 4T

dec  c                  ; 4T
jr   nz, .third_loop    ; 12T/7T


The `REPT 7` directive (supported by sjasmplus) repeats the block 7 times at assembly time --
row advance and third-advance happen at the fixed outer loop boundaries.

---

## Attribute Memory: 768 Bytes That Changed Everything

Below the pixel data, at `$5800`--`$5AFF`, sits the attribute memory. It is 768 bytes -- one f

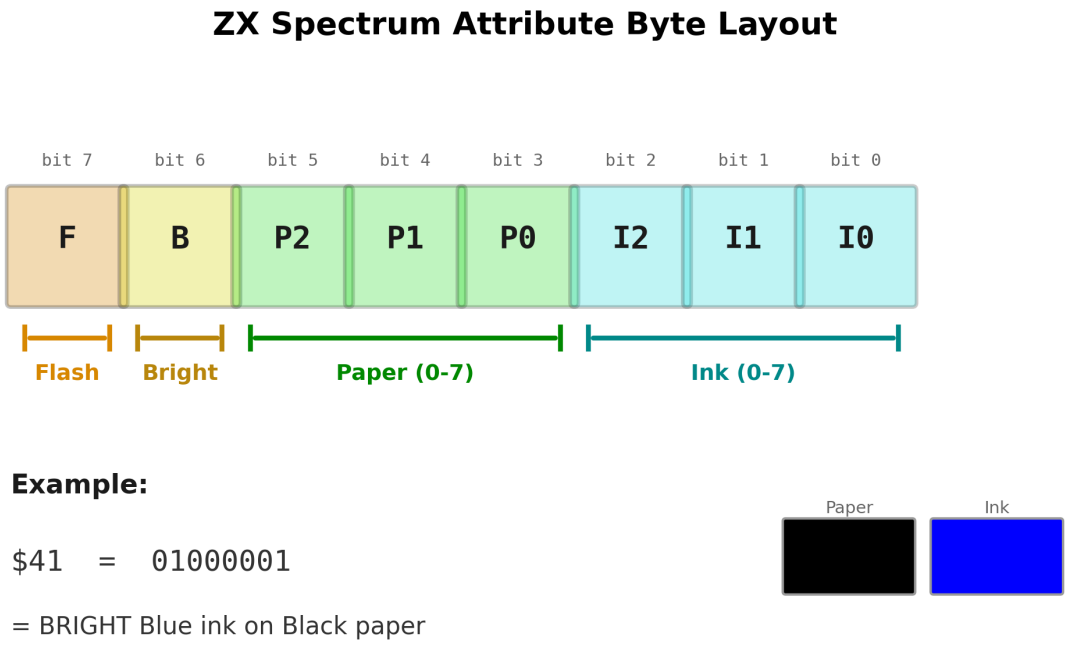
Each attribute byte has this layout:

```text
  Bit:   7     6     5  4  3     2  1  0
       +-----+-----+--------+--------+
       |  F  |  B  | PAPER  |  INK   |
       +-----+-----+--------+--------+

  F       = Flash (0 = off, 1 = flashing at ~1.6 Hz)
  B       = Bright (0 = normal, 1 = bright)
  PAPER   = Background colour (0-7)
  INK     = Foreground colour (0-7)
```

The 3-bit colour codes map to:

```
  0 = Black       4 = Green
  1 = Blue        5 = Cyan
  2 = Red         6 = Yellow
  3 = Magenta     7 = White
```

With the BRIGHT bit, each colour has a normal and bright variant. Black stays black whether bright or not, so the total palette is 15 distinct colours:

```
Normal:  Black  Blue  Red  Magenta  Green  Cyan  Yellow  White
Bright:  Black  Blue  Red  Magenta  Green  Cyan  Yellow  White
               (brighter versions of each)
```
```



Figure 4: Attribute byte bit layout showing flash, bright, paper, and ink fields

An attribute byte of `$47` = `01000111`: flash off (bit 7 = 0), bright **on** (bit 6 = 1), paper = 000 (black), ink = 111 (white). Bright white text on a black background. The non-bright version is `$07` = `00000111` – the Spectrum's default after `BORDER 0: PAPER 0: INK 7`.

This kind of bit-level detail matters when you are constructing attribute values at speed. A common pattern:

```
z80 id:ch02_attribute_memory_768_bytes_4 ; Build an attribute byte: bright white ink on blue paper ; Bright = 1, Paper = 001 (blue), Ink = 111 (white) ; = 01 001 111 = $4F     ld   a, $4F
```

The Attribute Clash

Here is the defining constraint of the ZX Spectrum: within each 8x8 pixel cell, you can only have **two colours** – ink and paper. Every set pixel (1) displays in the ink colour. Every clear pixel (0) displays in the paper colour. You cannot have three colours, or gradients, or per-pixel colouring, within a single cell.

This means that if a red sprite overlaps with a green background, the 8x8 cell containing the overlap must choose: all set pixels in this cell are either red or green. You cannot have some red and some green set pixels in the same cell. The visual result is a jarring block of colour that "clashes" with its surroundings – the infamous attribute clash.

```
Without clash (hypothetical per-pixel colour):

  +---------+---------+
  |  Red    | Red on  |
```

```
  |  sprite | green   |
  |  pixels | back-   |
  |         | ground  |
  +---------+---------+

With attribute clash (Spectrum reality):

  +---------+---------+
  |  Red    | Either  |
  |  sprite | ALL red |
  |  pixels | or ALL  |
  |         | green   |
  +---------+---------+

  The overlapping cell cannot have both colours.
```

Many early Spectrum games simply avoided the problem: monochrome graphics, or characters carefully designed to align with the 8x8 grid. Games like Knight Lore and Head Over Heels used a single ink/paper pair for the entire play area, eliminating clash entirely at the cost of colour.

But the demoscene saw this differently. Attribute clash is not just a limitation – it is a **creative constraint**. The 8x8 grid forces a particular aesthetic: bold blocks of colour, sharp geometric patterns, deliberate use of contrast. Demo effects that work entirely in attribute space – tunnels, plasmas, scrollers – can update 768 bytes per frame instead of 6,144, freeing enormous amounts of cycle budget for computation. When your entire display is attribute-driven, clash becomes irrelevant because you are not mixing sprites with backgrounds – the attributes *are* the graphics.

Introspec's demo Eager (2015) built its visual language entirely around this insight. The tunnel effect, the chaos zoomer, and the colour cycling animation all operate on attributes, not pixels. The result is an effect that runs at full frame rate with room to spare for digital drums and a sophisticated scripting engine. Clash is not a problem because the constraint was embraced from the start.

The Border: More Than Decoration

The 256x192 pixel display area sits in the centre of the screen, surrounded by a wide border. The border colour is set by writing to port `$FE`:

```
z80 id:ch02_the_border_more_than     ld   a, 1          ; 7T   blue = colour 1
out  ($FE), a       ; 11T  set border colour
```

Only bits 0–2 of the byte written to `$FE` affect the border colour. There are 8 colours (0–7), with no bright variants – the border palette is the non-bright set. Bits 3 and 4 of port `$FE` control the MIC and EAR outputs (tape interface and beeper sound), so you should mask or set those bits appropriately if you are not intending to make noise.

The border colour change takes effect immediately – on the very next scanline being drawn. This is what makes the border so useful as a debugging tool. As we saw in Chapter 1, changing the border colour before and after a section of code

creates a visible stripe whose height reveals the code's T-state cost. The border is your oscilloscope.

Border Effects

Because border colour changes are visible on the next scanline, precisely timed `OUT` instructions can create multicolour stripes, raster bars, and even crude graphics in the border area.

The basic principle: the ULA draws one scanline every 224 T-states (on Pentagon). If you execute an `OUT ($FE), A` instruction at the right moment, you change the border colour at a specific horizontal position on the current scanline. By executing a rapid sequence of `OUT` instructions with different colour values, you can paint horizontal stripes of colour in the border.

"'z80 id:ch02_border_effects ; Simple border stripes ; Assumes we are synced to the start of a border scanline

```
ld   a, 2          ; 7T   red
out  ($FE), a       ; 11T
; ... delay to fill this scanline ...
ld   a, 5          ; 7T   cyan
out  ($FE), a       ; 11T
; ... delay to fill next scanline ...
ld   a, 6          ; 7T   yellow
out  ($FE), a       ; 11T


More advanced border effects can create gradient bars, scrolling text, or even low-
resolution images. The challenge is extreme: you have 224 T-states per scanline, and each colo
states (7 for `LD A,n` + 11 for `OUT`). That gives you roughly 12 colour changes per scanline,

Demo coders have pushed this to remarkable extremes. By pre-loading multiple registers with co

For our purposes, the border's most important role is the one from Chapter 1: a free, always-
available timing visualiser. When you are optimising the screen-fill routine later in this cha

---

## Practical: The Checkerboard Fill

The example at `chapters/ch02-screen-as-puzzle/examples/fill_screen.a80` fills the pixel area

```z80 id:ch02_practical_the_checkerboard
    ORG $8000

SCREEN  EQU $4000       ; pixel area start
ATTRS   EQU $5800       ; attribute area start
SCRLEN  EQU 6144        ; pixel bytes (256*192/8)
ATTLEN  EQU 768         ; attribute bytes (32*24)
```

The code is placed at `$8000` – safely in uncontended memory on all Spectrum models. The constants name the key addresses and sizes.
```

```
z80 id:ch02_practical_the_checkerboard_2 start:     ; --- Fill pixels with
checkerboard pattern ---     ld   hl, SCREEN     ld   de, SCREEN + 1     ld
bc, SCRLEN - 1     ld   (hl), $55       ; checkerboard: 01010101     ldir
```

This uses the classic LDIR self-copy trick. It writes `$55` (binary `01010101`) to the first byte at `$4000`, then copies from each byte to the next for 6,143 bytes. The result: every byte of the pixel area is `$55`, which produces alternating set/clear pixels – a checkerboard. Because the pattern is the same in every byte, the interleaved row order does not matter – every row gets the same pattern regardless.

Cost: `LDIR` copies 6,143 bytes. The last iteration costs 16T, all others 21T: (6,143 - 1) x 21 + 16 = 128,998 T-states. Nearly two full frames on a Pentagon. This is fine for a one-time setup, but you would never do this in a per-frame rendering loop.

```
z80 id:ch02_practical_the_checkerboard_3     ; --- Fill attributes: white ink
on blue paper ---     ; Attribute byte: flash=0, bright=1, paper=001 (blue),
ink=111 (white)     ; = 01 001 111 = $4F     ld   hl, ATTRS     ld   de, ATTRS +
1     ld   bc, ATTLEN - 1     ld   (hl), $4F     ldir
```

Same technique for the attributes. The value `$4F` decodes as: flash off (0), bright on (1), paper blue (001), ink white (111). Every 8x8 cell gets bright white ink on blue paper. The checkerboard pixels are set/clear, so you see alternating white and blue dots – a classic ZX Spectrum visual pattern.

Cost: `LDIR` copies 767 bytes – (767 - 1) x 21 + 16 = 16,102 T-states.

“‘z80 id:ch02_practical_the_checkerboard_4 ; — Border: blue — ld a, 1 out ($FE), a

```
; Infinite loop
```

.wait: halt jr .wait

```
Sets the border to blue (colour 1) to match the paper colour, creating a visually clean frame.

### What to try

Load `fill_screen.a80` in your assembler and emulator. Then experiment:

- Change `$55` to `$AA` for the inverse checkerboard, or to `$FF` for solid fill, or `$81` for
- Change `$4F` to `$07` to see the same pattern without BRIGHT, or to `$38` for white paper wi
- Try `$C7` -- that sets the flash bit. Watch the characters alternate between ink and paper c
- Replace the LDIR pixel fill with a DOWN_HL loop that writes different patterns to different
7 (the first character cell's scanlines), the filled area will appear as 8 horizontal stripes

---

## Navigating the Screen: A Practical Summary

Here are the essential pointer operations for the Spectrum screen, collected in one place. The

### Moving right one byte (8 pixels)

```z80 id:ch02_moving_right_one_byte_8
    inc  l             ; 4T
```
```

This works within a character row because the column is in the low 5 bits of L. If you need to cross byte boundaries at the right edge (column 31 to column 0 of the next row), you need the full DOWN_HL plus reset of L – but typically you do not, because your loops are 32 bytes wide.

Moving down one pixel row

```
z80 id:ch02_moving_down_one_pixel_row     inc  h             ; 4T    (within a
character cell)
```

This works for 7 out of 8 rows. On the 8th row, you need the full boundary-crossing logic from the DOWN_HL routine above.

Moving down one character row (8 pixels)

```
z80 id:ch02_moving_down_one_character_row     ld   a, l          ; 4T     add
a, 32         ; 7T     ld   l, a          ; 4T    total: 15T (if no third crossing)
```

This advances by one character row within a third. If L overflows (carry set), you have crossed into the next third and need to add 8 to H.

Moving up one pixel row

```
z80 id:ch02_moving_up_one_pixel_row     dec  h             ; 4T    (within a
character cell)
```

The inverse of `INC H`. Same boundary issues at character cell and third boundaries. Here is the full UP_HL routine, the mirror of DOWN_HL:

“‘z80 id:ch02_moving_up_one_pixel_row_2 ; UP_HL: move HL one pixel row up on the Spectrum screen ; Input: HL = current screen address ; Output: HL = screen address one row above ; ; Classic version: up_hl: dec h ; 4T try moving one scan line up ld a, h ; 4T and 7 ; 7T did we cross a character boundary? cp 7 ; 7T ret nz ; 11/5T no: done

```
; Crossed a character cell boundary upward.
ld   a, l          ; 4T
sub  32            ; 7T   previous character row (L -= 32)
ld   l, a          ; 4T
ret  c             ; 11/5T  if carry, crossed into prev third

ld   a, h          ; 4T
add  a, 8          ; 7T   compensate H
ld   h, a          ; 4T
ret                ; 10T
```

There is a subtle optimisation here, contributed by Art-top (Artem Topchiy): replacing `and 7
states in the boundary-crossing path:

```z80 id:ch02_moving_up_one_pixel_row_3
; UP_HL optimised (Art-top)
; Saves 1 byte, 3 T-states on boundary crossing
;
```

```
up_hl_opt:
    dec  h              ; 4T
    ld   a, h           ; 4T
    cpl                 ; 4T   complement: 111 -> 000
    and  7              ; 7T   zero if we crossed boundary
    ret  nz             ; 11/5T

    ld   a, l           ; 4T
    sub  32             ; 7T
    ld   l, a           ; 4T
    ret  c              ; 11/5T

    ld   a, h           ; 4T
    add  a, 8           ; 7T
    ld   h, a           ; 4T
    ret                 ; 10T
```

The same `CPL / AND 7` trick works in DOWN_HL too, though the boundary condition there tests for `000` (which CPL turns into `111`, also non-zero after AND), so it does not help going downward. It is specifically the *upward* direction where the classic code needs the extra `CP 7` that the optimisation eliminates.

Computing the attribute address from a pixel address

If HL points to a byte in the pixel area, the corresponding attribute address can be calculated. Recall the pixel address structure: H = `010TTSSS`, L = `LLLCCCCC`. The attribute address for the same character cell is `$5800 + TT * 256 + LLL * 32 + CCCCC`. Since L already encodes `LLL * 32 + CCCCC` (which ranges 0–255), the attribute address is simply `($58 + TT) : L`. All we need to do is extract the two TT bits from H, combine them with `$58`, and leave L unchanged:

```
z80 id:ch02_computing_the_attribute ; Convert pixel address in HL to attribute
address in HL ; Input:  HL = pixel address ($4000-$57FF) ; Output: HL = corresponding
attribute address ($5800-$5AFF) ;     ld   a, h          ; 4T     rrca              ;
4T     rrca               ; 4T     rrca               ; 4T     and  3            ;
7T     or   $58           ; 7T     ld   h, a          ; 4T     ; L unchanged
--- Total: 34T
```

This works because L already contains `LLL CCCCC` – the character row within the third (0–7) combined with the column (0–31) – and that is exactly the low byte of the attribute address. The high byte just needs the third number added to `$58`. Elegant.

Special case: when H has scanline bits = 111. If you are iterating through a character cell top-to-bottom and have just processed the last scanline (scanline 7), the low 3 bits of H are `111`. In this case there is a faster 4-instruction conversion, contributed by Art-top:

```
z80 id:ch02_computing_the_attribute_2 ; Pixel-to-attribute when H low bits
are %111 ; (e.g., after processing the last scanline of a character cell) ;
Input:  HL where H = 010TT111 ; Output: HL = attribute address ;     srl  h
; 8T   010TT111 -> 0010TT11     rrc  h             ; 8T   0010TT11 -> 10010TT1
srl  h             ; 8T   10010TT1 -> 010010TT     set  4, h          ; 8T
010010TT -> 010110TT = $58+TT     ; L unchanged.     --- Total: 32T, 4 instructions
```

This is 2 T-states faster than the general method and avoids the `AND / OR` sequence. The trade-off is that it only works when the scanline bits are `111` – but that is exactly the situation after a top-to-bottom character cell render loop, which is one of the most common use cases.

Agon Light 2 Sidebar

The Agon Light 2's display is managed by a VDP (Video Display Processor) – an ESP32 microcontroller running the FabGL library. The eZ80 CPU communicates with the VDP over a serial link, sending commands to set graphics modes, draw pixels, define sprites, and manage palettes.

There is no interleaved memory layout. There is no attribute clash. The VDP supports multiple bitmap modes at various resolutions (from 640x480 down to 320x240 and below), with 64 colours or full RGBA palettes depending on the mode. Hardware sprites (up to 256) and tile maps are supported natively.

What changes for the programmer:

- **No address puzzle.** Pixel coordinates map linearly to buffer positions. You do not need DOWN_HL or split-counter screen traversal.
- **No attribute clash.** Each pixel can be any colour. The 8x8 grid constraint does not exist.
- **No direct memory access to the framebuffer.** The CPU cannot write directly to video memory the way a Spectrum CPU writes to `$4000`. Instead, you send VDP commands over the serial link. Drawing a pixel means sending a command sequence, not storing a byte. This introduces latency – the serial link runs at 1,152,000 baud – but it also means the CPU is free during rendering.
- **No cycle-level border tricks.** The VDP handles display timing independently. You cannot create raster effects by timing `OUT` instructions, because the display pipeline is decoupled from the CPU clock.

For a Spectrum programmer, the Agon feels freeing and frustrating in equal measure. The constraints that forced creative solutions on the Spectrum simply do not exist – but neither do the direct-hardware tricks that those constraints enabled. You trade the puzzle for an API.

Putting It Together: What the Screen Layout Means for Code

Every technique in the rest of this book is shaped by the screen layout described in this chapter. Here is why each piece matters:

Sprite drawing requires computing a screen address for the sprite's position, then iterating down through the sprite's rows. Each row means `INC H` (7 out of 8 times) or the full character-boundary crossing. A 16-pixel-tall sprite spans exactly 2 character cells – you will cross one boundary. A 24-pixel sprite spans 3 cells, crossing 2 boundaries. The boundary-crossing cost is a fixed tax on every sprite.

Screen clearing (Chapter 3) uses the PUSH trick – setting SP to `$5800` and pushing data downward through the pixel area. The interleave does not matter for clearing because every byte gets the same value. But for *patterned* clears (striped backgrounds, gradient fills), the interleave means you must think carefully about which rows get which data.

Scrolling (Chapter 17) is where the layout hurts most. Scrolling the screen up by one pixel means moving each row's 32 bytes to the address of the row above it. On a linear framebuffer, this is one big block copy. On the Spectrum, the source and destination addresses for each row are related by the DOWN_HL logic – not by a fixed offset. A scroll routine must navigate the interleave for every row it copies.

Attribute effects (Chapters 8–9) are where the layout helps. Because the attribute area is linear and small (768 bytes), updating colours is fast. A full-screen attribute update with LDIR costs about 16,000 T-states – less than a quarter of a frame. This is why attribute-based effects (tunnels, plasmas, colour cycling) are a staple of Spectrum demoscene work.

Summary

- The Spectrum's 6,912-byte display consists of **6,144 bytes of pixel data** at `$4000–$57FF` and **768 bytes of attributes** at `$5800–$5AFF`.
- Pixel rows are **interleaved** by character cell: the address encodes y as `010 TT SSS` (high byte) and `LLL CCCCC` (low byte), where the bits of y are shuffled across the address.
- Moving **one pixel row down** within a character cell is just `INC H` (4 T-states). Crossing character and third boundaries requires additional logic.
- The classic **DOWN_HL** routine handles all cases but costs up to 65 T-states at boundaries. For full-screen iteration, **split-counter loops** (Introspec's approach) reduce total cost by 60% and eliminate timing jitter.
- Each attribute byte encodes **Flash, Bright, Paper, and Ink** in the format `FBPPPIII`. Only **two colours per 8x8 cell** – this is the attribute clash.
- Attribute clash is not just a limitation but a **creative constraint** that defined the Spectrum's visual aesthetic and led to efficient attribute-only demo effects.
- The **border** colour is set by `OUT ($FE), A` (bits 0–2) and changes are visible on the next scanline, making it a **timing debug tool** and a canvas for demoscene raster effects.
- The **Agon Light 2** has no interleaved layout, no attribute clash, and no direct framebuffer access – it replaces the puzzle with a VDP command API.

Try It Yourself

1. **Map the addresses.** Pick 10 random (x, y) coordinates and calculate the screen address by hand using the `010TTSSS LLLCCCCC` bit layout. Then write a small Z80 routine that plots a single pixel at each coordinate and verify your calculations match.

2. **Visualise the interleave.** Modify `fill_screen.a80` to write different values

to the first 8 rows. Write `$FF` (solid) to row 0 and `$00` (empty) to rows 1–7. Because rows 0–7 are at `$4000`, `$4100`, …, `$4700`, you will need to change H to reach each row. The result should be a single bright line at the very top, with a gap of 7 empty lines before the next solid line at row 8.

3. **Time DOWN_HL.** Use the border-colour timing harness from Chapter 1. Call the classic DOWN_HL routine 191 times (for a full screen traversal) and measure the stripe. Then implement the split-counter version and compare. The split-counter version should produce a visibly shorter stripe.

4. **Attribute painter.** Write a routine that fills the attribute area with a gradient: column 0 gets colour 0, column 1 gets colour 1, and so on (cycling through 0–7). Each row should have the same pattern. Then modify it so each row shifts the pattern by one position – a diagonal rainbow. This is the seed of an attribute-based demo effect.

5. **Border stripes.** After a `HALT`, execute a tight loop that changes the border colour on every scanline for 64 lines. Use the 8 border colours in sequence (0, 1, 2, 3, 4, 5, 6, 7, repeat). You should see horizontal rainbow stripes in the top border. Adjust the timing delay between `OUT` instructions until the stripes are clean and stable.

Sources: Introspec “Eshchyo raz pro DOWN_HL” (Hype, 2020); Introspec “GO WEST Part 1” (Hype, 2015) for contended memory effects at screen addresses; Introspec “Making of Eager” (Hype, 2015) for attribute-based effect design; the Spectrum’s ULA documentation for memory layout rationale; Art-top (personal communication, 2026) for the optimised UP_HL and fast pixel-to-attribute conversion.

Next: Chapter 3 – The Demoscener’s Toolbox. Unrolled loops, self-modifying code, the stack as a data pipe, and the techniques that let you do the impossible within the budget.

Chapter 3: The Demoscener's Toolbox

Every craft has its bag of tricks — patterns that practitioners reach for so instinctively they stop thinking of them as tricks at all. A Z80 demoscener reaches for the techniques in this chapter.

These patterns — unrolled loops, self-modifying code, the stack as a data pipe, LDI chains, code generation, and RET-chaining — appear in almost every effect we will build in Part II. They are what separates a demo that fits in one frame from one that takes three. Learn them here, and you will recognise them everywhere.

Unrolled Loops and Self-Modifying Code

The cost of looping

Consider the simplest possible inner loop: clearing 256 bytes of memory.

```
z80 id:ch03_the_cost_of_looping ; Looped version: clear 256 bytes at (HL)     ld
b, 0            ; 7 T   (B=0 means 256 iterations)     xor  a               ; 4
T .loop:     ld   (hl), a         ; 7 T     inc  hl             ; 6 T     djnz
.loop           ; 13 T  (8 on last iteration)
```

Each iteration costs 7 + 6 + 13 = 26 T-states to store a single byte. Only 7 of those T-states do the work — the rest is overhead. That is 73% waste. For 256 bytes: 256 x 26 - 5 = 6,651 T-states. On a machine where you have 71,680 T-states per frame, those wasted T-states hurt.

Unrolling: trade ROM for speed

The solution is brutal and effective: write out the loop body N times and delete the loop.

```
z80 id:ch03_unrolling_trade_rom_for_speed ; Unrolled version: clear 256 bytes
at (HL)     xor  a               ; 4 T     ld   (hl), a         ; 7 T     inc
hl              ; 6 T     ld   (hl), a         ; 7 T     inc  hl             ;
6 T     ld   (hl), a         ; 7 T     inc  hl             ; 6 T     ; ...
repeated 256 times total
```

Each byte now costs 7 + 6 = 13 T-states. No DJNZ. No loop counter. Total: 256 x 13 = 3,328 T-states — half the looped version.

The cost is code size: 256 repetitions occupy 512 bytes vs. 7 for the loop. You are trading ROM for speed.

When to unroll: Inner loops that execute thousands of times per frame — screen clearing, sprite drawing, data copying.

When NOT to unroll: Outer loops that run once or twice per frame. Saving 5 T-states across 24 iterations buys you 120 T-states — less than three NOPs. Not worth the bloat.

The practical middle ground is *partial unrolling*: unroll 8 or 16 iterations inside the loop, keep DJNZ for the outer count. The `push_fill.a80` example in this chapter's `examples/` directory does exactly this: 16 PUSHes per iteration, 192 iterations.

Self-modifying code: the Z80's secret weapon

The Z80 has no instruction cache, no prefetch buffer, no pipeline. When the CPU fetches an instruction byte from RAM, it reads whatever is there *right now*. If you changed that byte one cycle ago, the CPU sees the new value. This is a guaranteed property of the architecture.

Self-modifying code (SMC) means writing to instruction bytes at runtime. The classic pattern is patching an immediate operand:

```
z80 id:ch03_self_modifying_code_the_z80_s ; Self-modifying code: fill with a
runtime-determined value     ld   a, (fill_value)       ; load the fill byte
from somewhere     ld   (patch + 1), a        ; overwrite the operand of the LD
below patch:     ld   (hl), $00             ; this $00 gets replaced at runtime
inc  hl     ; ...
```

The `ld (patch + 1), a` writes into the immediate operand of the next `ld (hl), $00`, changing it to `ld (hl), $AA` or whatever you loaded. The CPU executes whatever bytes it finds. Some common SMC patterns:

Patching opcodes. You can even replace the instruction itself. Need a loop that sometimes increments HL and sometimes decrements it? Before the loop, write the opcode for INC HL ($23) or DEC HL ($2B) into the instruction byte. Inside the inner loop, there is no branch at all — the right instruction is already in place. Compare this to a branch-per-iteration approach that would cost 12 T-states (JR NZ) on every single pixel.

Saving and restoring the stack pointer. This pattern appears constantly when using PUSH tricks (below):

```
z80 id:ch03_self_modifying_code_the_z80_s_2     ld   (restore_sp + 1), sp     ;
save SP into the operand below     ; ... do stack tricks ... restore_sp:     ld
sp, $0000                ; self-modified: the $0000 was overwritten
```

The `ld (nn), sp` saves the current SP directly into the operand of the later `ld sp, nn`. No temporary variable. This is idiomatic Z80 demoscene code.

Self-modifying variables: the `$+1` pattern

The most pervasive SMC pattern on the ZX Spectrum is not patching opcodes or saving SP — it is embedding a *variable* directly inside an instruction's immediate operand. The idea is simple: instead of storing a counter in a named memory

location and loading it with `LD A,(nn)` at 13 T-states, you let the instruction's own operand byte *be* the variable.

```
z80 id:ch03_smc_dollar_plus_one .smc_counter:     ld   a, 0                    ;
7T — this 0 is the "variable"     inc  a                        ; 4T     ld
(.smc_counter + 1), a   ; 13T — write back to the operand byte
```

The `ld a, 0` fetches its operand as part of the normal instruction decode — 7 T-states total, and the value is already in A. Compare that to loading from a separate memory address: `ld a, (counter)` costs 13 T-states, plus you still need a separate `ld (counter), a` at 13 T-states to write it back. The SMC version reads the variable for free (it is part of the instruction fetch) and only pays the 13 T-states once for the write-back.

In sjasmplus, you can place a label at `$+1` to give the embedded variable a readable name:

```
z80 id:ch03_smc_named_variable     ld   a, 0                    ; 7T .scroll_pos
EQU $ - 1           ; .scroll_pos names the operand byte above     add  a, 4
; 7T — advance by 4 pixels     ld   (.scroll_pos), a       ; 13T — store back
into the operand
```

This pattern appears everywhere in ZX Spectrum code: scroll positions, animation frame counters, effect phase accumulators, direction flags. Any single-byte value that persists between calls is a candidate. You will see it constantly in Parts II and V — practically every effect routine in this book uses at least one self-modifying variable.

The convention is to prefix these labels with `.smc_` or place them immediately after the instruction they modify. Either way, the intent should be clear to anyone reading the source. As we noted in Chapter 2, local labels (`.label`) prevent naming collisions when multiple routines each have their own embedded variables.

A word of caution. SMC is safe on the Z80, the eZ80, and every Spectrum clone. It is *not* safe on modern cached CPUs (x86, ARM) without explicit cache flush instructions. If you port to a different architecture, this is the first thing that breaks.

The Stack as a Data Pipe

Why PUSH is the fastest write on the Z80

The PUSH instruction writes 2 bytes to memory and decrements SP, all in 11 T-states. Let us compare the alternatives for writing data to a screen address:

Method	Bytes written	T-states	T-states per byte
`ld (hl), a` + `inc hl`	1	13	13.0
`ld (hl), a` + `inc l`	1	11	11.0
`ldi`	1	16	16.0
`ldir` (per byte)	1	21	21.0
`push hl`	2	11	**5.5**

PUSH writes two bytes in 11 T-states — 5.5 T-states per byte. Nearly 4x faster than LDIR. The catch: PUSH writes to where SP points, and SP is normally your stack. To use PUSH as a data pipe, you must hijack the stack pointer.

The technique

The pattern is always the same:

1. Disable interrupts (DI). If an interrupt fires while SP points to the screen, the CPU will push the return address into your pixel data. Chaos follows.
2. Save SP. Use self-modifying code to stash it.
3. Set SP to the *end* of your target area. The stack grows downward — PUSH decrements SP before writing. So if you want to fill from $4000 to $57FF, you set SP to $5800.
4. Load your data into register pairs and PUSH repeatedly.
5. Restore SP and re-enable interrupts (EI).

“‘mermaid id:ch03_the_technique graph TD A[“DI — disable interrupts”] –> B[“Save SP via self-modifying code”] B –> C[“Set SP to screen bottom ($5800)”] C –> D[“Load register pairs with fill data”] D –> E[“PUSH loop: 16 PUSHes per iteration11T × 16 = 176T → 32 bytes”] E –> F{All 192iterations done?} F – No –> E F – Yes –> G[“Restore SP from self-modified LD SP,nn”] G –> H[“EI — re-enable interrupts”]

```
style E fill:#f9f,stroke:#333
style A fill:#fdd,stroke:#333
style H fill:#fdd,stroke:#333


> **Why PUSH wins:** `LD (HL),A` + `INC HL` writes 1 byte in 13T (13.0 T/byte). `PUSH HL` writ

Here is the core of the `push_fill.a80` example from this chapter's `examples/` directory:

```z80 id:ch03_the_technique_2
stack_fill:
    di                          ; critical: no interrupts while SP is moved
    ld   (restore_sp + 1), sp   ; self-modifying: save SP

    ld   sp, SCREEN_END         ; SP points to end of screen ($5800)
    ld   hl, $AAAA              ; pattern to fill

    ld   b, 192                 ; 192 iterations x 16 PUSHes x 2 bytes = 6144
.loop:
    push hl                     ; 11 T  \
    push hl                     ; 11 T   |
    push hl                     ; 11 T   |
    push hl                     ; 11 T   |
    push hl                     ; 11 T   |
    push hl                     ; 11 T   |  16 PUSHes = 32 bytes
    push hl                     ; 11 T   |  = 176 T-states
    push hl                     ; 11 T   |
    push hl                     ; 11 T   |
    push hl                     ; 11 T   |
```
```

```
    push hl                     ; 11 T   |
    push hl                     ; 11 T   |
    push hl                     ; 11 T   |
    push hl                     ; 11 T   |
    push hl                     ; 11 T   |
    push hl                     ; 11 T  /
    djnz .loop                  ; 13 T (8 on last)

restore_sp:
    ld   sp, $0000              ; self-modified: restores original SP
    ei
    ret
```

The 16-PUSH inner body writes 32 bytes in 176 T-states. Total for the full 6,144-byte pixel area: roughly 36,000 T-states. Compare LDIR: 6,144 x 21 - 5 = 129,019 T-states. The PUSH method is about 3.6x faster — the difference between fitting in one frame and bleeding into the next.

POP as a fast read

PUSH is the fastest write, but POP is the fastest *read*. POP loads 2 bytes from (SP) into a register pair in 10 T-states — that is 5.0 T-states per byte. Compare the alternatives:

Method	Bytes read	T-states	T-states per byte
`ld a, (hl)` + `inc hl`	1	13	13.0
`ld a, (hl)` + `inc l`	1	11	11.0
`ldi` (as a read+write)	1	16	16.0
`pop hl`	2	10	**5.0**

The pattern: pre-build a table of 16-bit values in memory, point SP at the start of the table, and POP into register pairs. Each POP advances SP by 2, walking through the table automatically. This is the read-side complement of the PUSH write trick.

Combine POP and PUSH and you get a fast memory-to-memory pipe: POP a value from a source table (10T), process the register pair if needed, then PUSH it to the destination (11T). Total: 21 T-states for 2 bytes — the same throughput as LDIR, but with the register pair available for processing between the read and write. You can mask bits, add offsets, swap bytes, or apply any register-to-register transformation at no extra memory-access cost. This POP-process-PUSH pipeline is the backbone of many compiled sprite routines.

Where PUSH tricks are used

- **Screen clearing.** The most common use. Every demo needs to clear the screen between effects.
- **Compiled sprites.** The sprite is compiled into a sequence of PUSH instructions with pre-loaded register pairs. The fastest possible sprite output on the Z80.

- **Fast data output.** Any time you need to blast a block of data to a contiguous address range: attribute fills, buffer copies, display list construction.

The price you pay: interrupts are off. If your music player runs from an IM2 interrupt, it will miss a beat during a long PUSH sequence. Demo coders plan around this — schedule PUSH fills during border time, or split them across multiple frames.

LDI Chains

LDI vs LDIR

LDI copies one byte from (HL) to (DE), increments both, and decrements BC. LDIR does the same but repeats until BC = 0. The difference is timing:

Instruction	T-states	Notes
LDI	16	Copies 1 byte, always 16 T
LDIR (per byte)	21	Copies 1 byte, loops back. Last byte: 16 T

LDIR costs 5 extra T-states per byte for its internal loop-back check. Those 5 T-states add up fast.

For 256 bytes: - LDIR: 255 x 21 + 16 = 5,371 T-states - 256 x LDI: 256 x 16 = 4,096 T-states - Savings: 1,275 T-states (24%)

A chain of individual LDI instructions is just 256 repetitions of the two-byte opcode `$ED $A0`. That is 512 bytes of code to save 24% — the same ROM-for-speed trade-off as loop unrolling.

When LDI chains shine

The sweet spot is copying blocks of known size. A chain of 32 LDIs saves 160 T-states over LDIR for a sprite row. Across 24 rows, that is 3,840 T-states per frame.

But the real power emerges when you combine LDI chains with *entry point arithmetic*. If you have a chain of 256 LDIs and want to copy only 100 bytes, jump into the chain at position 156. No loop counter, no setup. This technique is used in Introspec's chaos zoomer in Eager (2015):

"'z80 id:ch03_when_ldi_chains_shine ; Chaos zoomer inner loop (simplified from Eager) ; Each line copies a different number of bytes from a source buffer. ; Entry point into the LDI chain is calculated per line. ld hl, source_data ld de, dest_screen ; ... calculate entry point based on zoom factor ... jp (ix) ; jump into the LDI chain at the right point

ldi_chain: ldi ; byte 255 ldi ; byte 254 ldi ; byte 253 ; ... 256 LDIs total ... ldi ; byte 0 ; falls through to next line setup

```
This variable-length copy with zero per-byte loop overhead is a technique you simply cannot ac
```

```
## Bit Tricks: SBC A,A and Friends

### SBC A,A as a conditional mask

After any instruction that produces a carry flag, `SBC A,A` converts that flag into a full byt
states. Compare this to the branching alternative --- `JR C,.set` / `LD A,0` / `JR .done` / `.
22 T-states depending on which path is taken, plus the pipeline disruption of a conditional br

The canonical use case is *bit-to-byte expansion*. Given a byte where each bit represents a pi

```z80 id:ch03_sbc_bit_expand
    rlc  (hl)           ; rotate top bit into carry    — 15T
    sbc  a, a           ; A = $FF if set, $00 if not   — 4T
    and  $47            ; A = bright white ($47) or $00 — 7T
```

Three instructions, 26 T-states, no branches. To select between two *arbitrary* values rather than zero and a mask, use the pattern `SBC A,A : AND mask : XOR base`. The AND selects which bits change between the two values, and the XOR flips them to the desired base. This pattern replaces every "if bit set then value A else value B" test in your inner loops.

ADD A,A vs SLA A

Both instructions shift A left by one bit. But `ADD A,A` is 4 T-states and 1 byte, while `SLA A` is 8 T-states and 2 bytes. There is no situation where SLA A is preferable — `ADD A,A` is strictly faster and smaller. Similarly, `ADD HL,HL` shifts HL left in 11 T-states (1 byte), replacing the two-instruction sequence `SLA L : RL H` at 16 T-states (4 bytes). For a 16-bit left shift inside an inner loop running 192 times per frame, that substitution alone saves 960 T-states — over four scanlines of border time.

These are not tricks. They are vocabulary. Just as a fluent speaker does not pause to conjugate common verbs, a Z80 programmer reaches for `ADD A,A` and `SBC A,A` without conscious thought. If you find yourself writing `SLA A` or a conditional branch to select between two values, stop and reach for the shorter form. The T-states add up.

Code Generation

Code generation: writing the program that draws

Everything above is a fixed optimisation — the code runs the same way every frame. Code generation goes further: your program writes the program that draws the screen. There are two variants: offline (before assembly) and runtime (during execution).

Offline: generating assembly from a higher-level language

Introspec used Processing (a Java-based creative coding environment) to generate Z80 assembly for the chaos zoomer in Eager (2015). A chaos zoomer changes
```

magnification every frame — different source pixels map to different screen locations. Rather than computing these mappings at runtime, the Processing script pre-calculated every mapping and output .a80 source files containing optimised LDI chains and LD instructions.

The workflow: a Processing script calculates, for each frame, which source byte maps to which screen byte. It outputs Z80 assembly source — sequences of `ld hl, source_addr` and `ldi` instructions — which the assembler (sjasmplus) builds alongside the hand-written engine code. At runtime, the engine simply calls into the pre-generated code for the current frame.

This is not "cheating." It is the fundamental insight that division of labour between compile time and runtime can eliminate branches, lookups, and arithmetic from the inner loop entirely. The Processing script does the hard maths once, slowly, on a modern machine. The Z80 does the easy part — copying bytes — as fast as physically possible.

Runtime: the program writes machine code during execution

Sometimes parameters change every frame, so offline generation is not enough. The sphere-mapping routine in X-Trade's Illusion (ENLiGHT'96) generates machine code into a RAM buffer at runtime. The sphere geometry changes as it rotates — different pixels need different skip distances. Before each frame, the engine emits opcode bytes into a buffer, then executes them:

"'z80 id:ch03_runtime_the_program_writes ; Runtime code generation (conceptual, simplified from Illusion) ; Generate an unrolled rendering loop for this frame's sphere slice

```
ld   hl, code_buffer
ld   de, sphere_table       ; per-frame skip distances

ld   b, SPHERE_WIDTH
```

.gen_loop: ld a, (de) ; load skip distance for this pixel inc de

```
; Emit: ld a, (hl) -- opcode $7E
ld   (hl), $7E
inc  hl

; Emit: add a, N   -- opcodes $C6, N
ld   (hl), $C6
inc  hl
ld   (hl), a                ; the skip distance, as immediate operand
inc  hl

djnz .gen_loop

; Emit: ret -- opcode $C9
ld   (hl), $C9

; Now execute the generated code
call code_buffer
```

```
The generated code is a straight-line sequence with no branches, no lookups, no loop overhead
states per branch, you emit the exact instructions needed and execute them branch-
free.

### The cost of generation

Runtime code generation is not free. Look at the generator loop above: each emitted instructio
50 T-states per emitted byte, depending on complexity. Call it ~40 T-states on average. For a
states of generation overhead.

The break-even point: generation pays off when the generated code runs more than once per fram
pixel conditional branches that would cost far more. Alone Coder documented a similar trade-
off in his rotation engine: generating a sequence of INC H/INC L instructions for coordinate s
states to emit, but eliminates coordinate arithmetic that would cost approximately 146,000 T-
states if computed inline. The generation overhead is under 4% of the cost it replaces.

The rule of thumb: if you find yourself writing a loop that contains branches selecting betwee
pixel or per-line data, that loop is a candidate for code generation. Emit the right instructi
free, and regenerate only when the parameters change.

**When to generate code:** If the same operations happen every frame with only data changes, s
modifying code (patching operands) suffices. If the *structure* changes --- different numbers

---

## RET-Chaining

### Turning the stack into a dispatch table

In 2025, DenisGrachev published a technique on Hype developed for his game Dice Legends. The p
based playfield requires drawing dozens of tiles per frame. The naive approach uses CALL:

```z80 id:ch03_turning_the_stack_into_a
; Naive approach: call each tile renderer
    call draw_tile_0
    call draw_tile_1
    call draw_tile_2
    ; ...
```

Each CALL costs 17 T-states. For a 30 x 18 playfield (540 tiles), that is 9,180 T-states on dispatch alone.

DenisGrachev's insight: set SP to a *render list* — a table of addresses — and end each tile-drawing procedure with RET. RET pops 2 bytes from (SP) into PC. If SP points to your render list, RET does not return to the caller — it jumps to the next routine in the list.

"'z80 id:ch03_turning_the_stack_into_a_2 ; RET-chaining: zero call overhead di ld (restore_sp + 1), sp ; save SP ld sp, render_list ; SP points to our dispatch table

```
; "Call" the first tile routine by falling into it or using RET:
ret                         ; pops first address from render_list
```

; Each tile routine ends with: draw_tile_N: ; ... draw the tile ... ret ; pops NEXT
```

address from render_list

; The render list is a sequence of addresses: render_list: dw draw_tile_42 ; first tile to draw dw draw_tile_7 ; second tile dw draw_tile_42 ; third tile (same tile type, different position) ; ... one entry per tile on screen ... dw render_done ; sentinel: address of cleanup code

render_done: restore_sp: ld sp, $0000 ; self-modified: restore SP ei

```
Each dispatch now costs 10 T-states (RET) instead of 17 (CALL). For 540 tiles: 3,780 T-
states saved. But the real gain is free dispatch --- each entry can point to a different proce

### Three strategies for the render list

DenisGrachev explored three approaches to constructing the render list:

1. **Map as render list.** The tilemap itself is the render list: each cell contains the addre

2. **Address-based segments.** The screen is divided into segments. Each segment's render list

3. **Byte-based with 256-byte lookup tables.** Each tile type is a single byte (the tile index
byte lookup table maps tile indices to routine addresses. The render list is built by iteratin

Using the byte-based approach, he expanded the playfield from 26 x 15 tiles (the limit of his
cost dispatch, freed enough T-states to render 40% more tiles.

### The trade-offs

Like all stack tricks, interrupts must be disabled while SP is hijacked. Each tile routine mus
contained --- ending with RET and not using CALL, since the real stack is unavailable. In prac

---

## Sidebar: "Code is Dead" (Introspec, 2015)

In January 2015, Introspec published a short, provocative essay on Hype titled "Code is Dead"

The uncomfortable truth: modern demos are consumed as visual media. People watch them on YouTu
states per pixel is invisible to 99% of the audience. "Writing code purely for its own sake,"

And yet.

You are reading this book. We are opening the debugger. We are counting T-
states. We are looking inside. The techniques in this chapter are not museum exhibits. They ar

Introspec's essay is a challenge, not a surrender. He went on to publish some of the most deta

---

## Putting It All Together

The techniques in this chapter are not independent. In practice, they compose:
```

```
- **Screen clearing** combines *unrolled loops* with *PUSH tricks*: a partially unrolled loop
modifying code*.
- **Compiled sprites** combine *code generation* (each sprite compiles to executable code), *P
modification* (patching screen addresses per frame).
- **Tile engines** combine *RET-chaining* for dispatch with *LDI chains* inside each tile rout
- **Chaos zoomers** combine *offline code generation* (Processing scripts emitting assembly) w
modification* (patching source addresses per frame).
- **Attribute effects** combine *POP reads* from pre-computed tables with *bit tricks* (SBC A,

The common thread: every technique eliminates something from the inner loop. Unrolling elimina
modification eliminates branches. PUSH eliminates per-byte write overhead. POP eliminates per-
byte read overhead. LDI chains eliminate the LDIR repeat penalty. Bit tricks eliminate conditi
chaining eliminates CALL overhead.

The Z80 runs at 3.5 MHz. You have 71,680 T-states per frame. Every T-state you save in the inn
state you can spend on more pixels, more colours, more motion. The toolbox in this chapter is

In the chapters that follow, you will see each of these techniques at work in real demos --- t

---

## Try It Yourself

1. **Measure the difference.** Take the timing harness from Chapter 1 and measure three versio
byte fill: (a) the `ld (hl), a : inc hl : djnz` loop, (b) a fully unrolled `ld (hl), a : inc h
based fill from `examples/push_fill.a80`. Compare the border stripe widths. The PUSH version's

2. **Build a self-modifying clear.** Write a screen-clear routine that takes the fill pattern
based fill loop using self-modifying code. Call it twice with different patterns and watch the

3. **Time an LDI chain.** Write a 32-byte copy using LDIR and another using 32 x LDI. Measure
colour technique. The LDI chain should save 160 T-states --- visible if you run the copy in a

4. **Experiment with entry points.** Build a 128-entry LDI chain and a small routine that calc
128). Jump into the chain at different points. This is a simplified version of the variable-
length copy used in real chaos zoomers.

5. **Variable-length copier with calculated entry.** Build a 256-entry LDI chain and a front-
end that accepts a byte count in register B (1--256). Calculate the entry point: each LDI is 2
colour harness and compare the stripe width against LDIR for the same byte count. For small co

6. **Bit-to-attribute unpacker.** Write a routine that reads a byte from (HL), rotates each bi

> **Sources:** DenisGrachev "Tiles and RET" (Hype, 2025); Introspec "Making of Eager" (Hype, 2


\newpage

# Chapter 4: The Maths You Actually Need
```

```
> *"Read a maths textbook -- derivatives, integrals. You will need them."*
> -- Dark, Spectrum Expert #01 (1997)

In 1997, a teenager in St. Petersburg sat down to write a magazine article about multiplicatio
Trade, and his demo *Illusion* had already won first place at ENLiGHT'96. Now he was writing *

What follows is drawn directly from Dark's "Programming Algorithms" article in Spectrum Expert
engineered *Illusion* twenty years later on the Hype blog, he found these exact algorithms at

---

## Multiplication on Z80

The Z80 has no multiply instruction. Every time you need A times B -- for rotation matrices, p
off between them.

### Method 1: Shift-and-Add from LSB

The classic approach. Scan through the bits of the multiplier from LSB to MSB. For each set bi

Here is Dark's 8x8 unsigned multiply. Input: B times C. Result in A (high byte) and C (low byt

```z80 id:ch04_method_1_shift_and_add_from
; MULU112 -- 8x8 unsigned multiply
; Input:  B = multiplicand, C = multiplier
; Output: A:C = B * C (16-bit result, A=high, C=low)
; Cost:   196-204 T-states (Pentagon)
;
; From Dark / X-Trade, Spectrum Expert #01 (1997)

mulu112:
    ld   a, 0           ; clear accumulator (high byte of result)
    ld   d, 8           ; 8 bits to process

.loop:
    rr   c              ; shift LSB of multiplier into carry
    jr   nc, .noadd     ; if bit was 0, skip addition
    add  a, b           ; add multiplicand to accumulator
.noadd:
    rra                 ; shift accumulator right (carry into bit 7,
                        ;   bit 0 into carry -- this carry feeds
                        ;   back into C via the next RR C)
    dec  d
    jr   nz, .loop
    ret
```

Study this carefully. The `RRA` instruction shifts A right, but also pushes A's lowest bit into the carry flag. On the next iteration, `RR C` rotates that carry into the top of C. So the low bits of the product gradually assemble in C, while the high bits accumulate in A. After eight iterations, the full 16-bit result sits in A:C.

The cost is 196 to 204 T-states depending on how many multiplier bits are set –
```

each set bit costs one extra `ADD A,B` (4 T-states). The example at `chapters/ch04-maths/examples/multiply8.a80` shows a variant returning the result in HL.

For 16x16 producing a 32-bit result, Dark's MULU224 runs in 730 to 826 T-states. In practice, demoscene 3D engines avoid full 16x16 multiplies by keeping coordinates in 8.8 fixed-point and using 8x8 multiplies where possible.

Shift-and-Add Multiply: 13 x 11 = 143

Multiplier = 13 = 00001101 Multiplicand = 11 = 00001011

Step	Bit	Multiplier bits	Action	Acc	Acc (binary)
0	b0 = 1	0000110**1**	ADD 11	11	00001011
1	b1 = 0	000011**0**1	skip	11	00001011
2	b2 = 1	00001**1**01	ADD 44	55	00110111
3	b3 = 1	0000**1**101	ADD 88	143	10001111
4	b4 = 0	000**0**1101	skip	143	10001111
5	b5 = 0	00**0**01101	skip	143	10001111
6	b6 = 0	0**0**001101	skip	143	10001111
7	b7 = 0	**0**0001101	skip	143	10001111

Result: 13 x 11 = 143 ($8F = 10001111)

Bit set: ADD multiplicand Bit clear: skip (no addition)

Figure 5: Shift-and-add 8x8 multiply walkthrough

Method 2: Square Table Lookup

Dark's second method trades memory for speed, exploiting an algebraic identity that every demoscener eventually discovers:

```
A * B = ((A+B)^2 - (A-B)^2) / 4
```

Pre-compute a table of n^2/4 values, and multiplication becomes two lookups and a subtraction – approximately 61 T-states, more than three times faster than shift-and-add.

You need a 512-byte table of (n^2/4) for n = 0 to 511, page-aligned for single-register indexing. The table must be 512 bytes because (A+B) can reach 510.

“‘z80 id:ch04_method_2_square_table_lookup_2 ; MULU_FAST – Square table multiply ; Input: B, C = unsigned 8-bit factors ; Output: HL = B * C (16-bit result) ; Cost: ~61 T-states (Pentagon) ; Requires: sq_table = 512-byte table of n^2/4, page-aligned ; ; A*B = ((A+B)^2 - (A-B)^2) / 4

mulu_fast: ld h, sq_table » 8 ; high byte of table address ld a, b add a, c ; A = B + C (may overflow into carry) ld l, a ld e, (hl) ; look up (B+C)^2/4 low byte inc h ld d, (hl) ; look up (B+C)^2/4 high byte

```
ld   a, b
sub  c                 ; A = B - C (may go negative)
jr   nc, .pos
neg                    ; take absolute value
```

.pos: ld l, a dec h ld a, e sub (hl) ; subtract (B-C)^2/4 low byte ld e, a inc h ld a, d sbc a, (hl) ; subtract (B-C)^2/4 high byte ld d, a

```
ex   de, hl            ; HL = result
ret


The trade-off? Dark is characteristically honest: **"Choose: speed or accuracy."** The table s
and-add, the rotation is perfectly smooth.

For texture mapping, plasma, scrollers -- use the fast multiply. For wireframe 3D where the ey
and-add. Dark knew this because he had tried both in *Illusion*.

**Generating the square table** is a one-time startup cost. Dark suggests using the derivative

---

## Division on Z80

Division on the Z80 is even more painful than multiplication. No divide instruction, and the a

### Method 1: Shift-and-Subtract (Restoring Division)

Binary long division. Start with a zeroed accumulator. The dividend shifts in from the right,

```z80 id:ch04_method_1_shift_and_subtract
; DIVU111 -- 8-bit unsigned divide
; Input:  B = dividend, C = divisor
; Output: B = quotient, A = remainder
; Cost:   236-244 T-states (Pentagon)
;
; From Dark / X-Trade, Spectrum Expert #01 (1997)

divu111:
```
```

```
    xor  a              ; clear accumulator (remainder workspace)
    ld   d, 8           ; 8 bits to process

.loop:
    sla  b              ; shift dividend left -- MSB into carry
    rla                 ; shift carry into accumulator
    cp   c              ; try to subtract divisor
    jr   c, .too_small  ; if accumulator < divisor, skip
    sub  c              ; subtract divisor from accumulator
    inc  b              ; set bit 0 of quotient (B was just shifted,
                        ;   so bit 0 is free)
.too_small:
    dec  d
    jr   nz, .loop
    ret                 ; B = quotient, A = remainder
```

The `INC B` to set the quotient bit is a neat trick: B was just shifted left by `SLA B`, so bit 0 is guaranteed zero. `INC B` sets it without affecting other bits – cheaper than `OR` or `SET`.

The 16-bit version (DIVU222) costs 938 to 1034 T-states. A thousand T-states for a single divide. With a frame budget of ~70,000 T-states, you can afford perhaps 70 divides per frame – doing nothing else. This is why demoscene 3D engines go to extreme lengths to avoid division.

Method 2: Logarithmic Division

Dark's faster alternative uses logarithm tables:

```
Log(A / B) = Log(A) - Log(B)
A / B = AntiLog(Log(A) - Log(B))
```

With two 256-byte lookup tables – Log and AntiLog – division becomes two lookups, a subtraction, and a third lookup. Cost drops to roughly 50-70 T-states. For perspective division (dividing by Z to project 3D points onto screen), this is a game-changer.

Generating the log table is where things get interesting. Dark proposes building it using derivatives – the same incremental technique as the square table. The derivative of log2(x) is 1/(x * ln(2)), so you accumulate fractional increments step by step, starting from log2(1) = 0 and working upward. The constant 1/ln(2) = 1.4427 needs to be scaled to fit the table's 8-bit range.

And here is where Dark's honesty shines through. After deriving the generation formula, he attempts to compute a correction coefficient for the table scaling and arrives at 0.4606. He then writes – in a published magazine article – *"Something is not right here, so it is recommended to write a similar one yourself."*

A seventeen-year-old in 1997, publishing in a disk magazine read by his peers across the Russian Spectrum scene, openly saying: I got this working, but my derivation has a hole in it, figure out the clean version yourself. That honesty is rare in technical writing at any level, and it is one of the things that makes Spectrum Expert such a remarkable document.

In practice, the log tables work. Rounding errors from compressing a continuous

function into 256 bytes are acceptable for perspective projection. Dark's 3D engine in *Illusion* uses exactly this technique.

Sine and Cosine

Rotation, scrolling, plasma – every effect that curves needs trigonometry. On the Z80, you pre-compute a lookup table. Dark's approach is beautifully pragmatic: a parabola is close enough to a sine wave for demo work.

The Parabolic Approximation

Half a period of cosine, from 0 to pi, curves from +1 down to -1. A parabola y = 1 - 2*(x/pi)^2 follows almost the same path. Maximum error is about 5.6% – terrible for engineering, invisible in a demo at 256x192 resolution.

Dark generates a 256-byte signed cosine table (-128 to +127), indexed by angle: 0 = 0 degrees, 64 = 90 degrees, 128 = 180 degrees, 256 wraps to 0. The power-of-two period means the angle index wraps naturally with 8-bit overflow, and cosine becomes sine by adding 64.

"'z80 id:ch04_the_parabolic_approximation ; Generate 256-byte signed cosine table (-128..+127) ; using parabolic approximation ; ; The table covers one full period: cos(n * 2*pi/256) ; scaled to signed 8-bit range. ; ; Approach: for the first half (0..127), compute ; y = 127 - (x^2 255 / 128^2) ; approximated via incrementing differences. ; Mirror for second half.

gen_cos_table: ld hl, cos_table ld b, 0 ; x = 0 ld de, 0 ; running delta (fixed-point)

```
; First quarter: cos descends from +127 to 0
; Second quarter: continues to -128
; ...build via incremental squared differences

; In practice, the generation loop runs ~30 bytes
; and produces the table in a few hundred cycles.


The key insight: you do not need to compute x^2 for each entry. Since (x+1)^2 - x^2 = 2x + 1,

The resulting table is a piecewise parabolic approximation. Plot it against true sine and you

> **Sidebar: Raider's 9 Commandments of Sine Tables**
>
> In the Hype comments on Introspec's analysis of *Illusion*, veteran coder Raider dropped a l
>
> - Use a power-of-two table size (256 entries is canonical).
> - Align the table to a page boundary so `H` holds the base and `L` is the raw angle -- index
> - Store signed values for direct use in coordinate arithmetic.
> - Let the angle wrap naturally via 8-bit overflow -- no bounds checking.
> - Cosine is just sine offset by a quarter period: load angle, add 64, look up.
> - If you need higher precision, use a 16-bit table (512 bytes) but you rarely do.
> - Generate the table at startup rather than storing it in the binary -- saves space, costs n
> - For 3D rotation, pre-multiply by your scaling factor and store the scaled values.
```

```
> - Never compute trigonometry at runtime. If you think you need to, you are wrong.
>
> These commandments reflect decades of collective experience. Follow them and your sine table

---

## Bresenham's Line Drawing

Every edge of a wireframe object is a line from (x1,y1) to (x2,y2), and you need to draw it fa

### The Classic Algorithm and Xopha Modification

Bresenham's algorithm steps along the major axis one pixel at a time, maintaining an error acc
axis steps. On the Spectrum, "set a pixel" is expensive -- the interleaved screen memory means
states. The ROM routine takes over 1000 T-states per pixel. Even a hand-optimised Bresenham lo
states per pixel.

Dark mentions Xopha's improvement: maintain a screen pointer (HL) and advance it incrementally
instruction DOWN_HL adjustment. Better, but the core problem remains.

### Dark's Matrix Method: 8x8 Pixel Grids

Then Dark makes his key observation: **"87.5% of checks are wasted."**

In a Bresenham loop, at every pixel you ask: should I step sideways? For a nearly horizontal l
step. You are burning T-states on a conditional branch that almost never fires.

Dark's solution: pre-compute the pixel pattern for each line slope within an 8x8 pixel grid, a
pixel patterns as straight sequences of `SET bit,(HL)` instructions with address increments be

```z80 id:ch04_dark_s_matrix_method_8x8
; Example: one unrolled 8-pixel segment of a nearly-horizontal line
; (octant 0: moving right, gently sloping down)
;
; The line enters at the left edge of an 8x8 character cell
; and exits at the right edge, dropping one pixel row partway through.

    set  7, (hl)        ; pixel 0 (leftmost bit in byte)
    set  6, (hl)        ; pixel 1
    set  5, (hl)        ; pixel 2
    set  4, (hl)        ; pixel 3
    set  3, (hl)        ; pixel 4
    ; --- step down one pixel row ---
    inc  h              ; next screen row (within character cell)
    set  2, (hl)        ; pixel 5
    set  1, (hl)        ; pixel 6
    set  0, (hl)        ; pixel 7 (rightmost bit in byte)
```

No conditional branches. No error accumulator. `SET bit,(HL)` takes 15 T-states; eight of them plus a couple of `INC H` operations gives ~130 T-states per 8-pixel segment, or about 16 T-states per pixel. With lookup and cell-advance overhead, Dark achieves approximately **48 T-states per pixel** – nearly half the classical
```

Bresenham cost.

The price is memory: a separate unrolled routine for each slope per octant, about **3KB total**. On a 128K Spectrum, a modest investment for a massive speed gain.

Trap-Based Termination

Instead of checking a loop counter at every pixel, Dark plants a sentinel where the line ends. When the drawing code hits the sentinel, it exits – eliminating the `DEC counter / JR NZ` overhead entirely.

The complete system – octant selection, segment lookup, unrolled drawing, trap termination – is one of the most impressive pieces of code in Spectrum Expert #01. When Introspec disassembled *Illusion* in 2017, he found this matrix method at work, drawing the wireframes at full frame rate.

Fixed-Point Arithmetic

Every algorithm in this chapter assumes something we have not yet made explicit: fixed-point numbers.

The Z80 has no floating-point unit. Every register holds an integer. But demo effects need fractional values – rotation angles, sub-pixel velocities, scale factors. The solution is fixed-point: pick a convention for where the "decimal point" lives within an integer, then do all arithmetic in integers while tracking the scaling mentally.

Format 8.8

The most common format on the Z80 is **8.8**: high byte = integer part, low byte = fractional part. One 16-bit register pair holds one fixed-point number:

```
H = integer part    (-128..+127 signed, or 0..255 unsigned)
L = fractional part (0..255, representing 0/256 to 255/256)
```

`HL = $0180` represents 1.5 (H=1, L=128, and 128/256 = 0.5). `HL = $FF80` signed is -0.5 (H=$FF = -1 in two's complement, L=$80 adds 0.5).

The beauty: **addition and subtraction are free** – just normal 16-bit operations:

"'z80 id:ch04_format_8_8_2 ; Fixed-point 8.8 addition: result = a + b ; HL = first operand, DE = second operand add hl, de ; that's it. 11 T-states.

; Fixed-point 8.8 subtraction: result = a - b or a ; clear carry sbc hl, de ; 15 T-states.

```
The processor does not care that you are treating these as fixed-point. Binary addition is the

### Fixed-Point Multiplication

Multiplying two 8.8 numbers produces a 16.16 result -- 32 bits. You want 8.8 back, so you take
1 and +1), you can decompose the multiply into partial products:

```z80 id:ch04_fixed_point_multiplication
```
```

```
; Fixed-point 8.8 multiply (simplified)
; Input:  BC = first operand (B.C in 8.8)
;         DE = second operand (D.E in 8.8)
; Output: HL = result (H.L in 8.8)
;
; Full product = BC * DE (32 bits), we want bits 8..23
;
; Decomposition:
;   BC * DE = (B*256+C) * (D*256+E)
;           = B*D*65536 + (B*E + C*D)*256 + C*E
;
; In 8.8 result (bits 8..23):
;   H.L = B*D*256 + B*E + C*D + (C*E)/256
;
; For small B,D (say -1..+1), B*D*256 is the dominant term.
; C*E/256 is a rounding correction.
; Total cost: ~200 T-states using the shift-and-add multiplier.

fixmul88:
    ; Multiply B*E -> add to result high
    ld   a, b
    call mul8             ; A = B*E (assuming 8x8->8 truncated)
    ld   h, a

    ; Multiply C*D -> add to result
    ld   a, c
    ld   b, d
    call mul8             ; A = C*D
    add  a, h
    ld   h, a

    ; For higher precision, also compute B*D and C*E
    ; and combine. In practice, the two middle terms
    ; are often sufficient for demo work.

    ld   l, 0             ; fractional part (approximate)
    ret
```

For sine-table-driven rotation where sine values are 8-bit signed (-128 to +127, representing -1.0 to +0.996), multiplying an 8-bit coordinate by a sine value via `mulu112` gives a 16-bit result already in 8.8 format – high byte is the rotated integer coordinate, low byte is the fraction.

Why Fixed-Point Matters

Format 8.8 is the sweet spot for the Z80: fits in a register pair, add/subtract are free, multiply costs ~200 T-states, and precision is sufficient for screen-resolution effects. Other formats exist – 4.12 for more fractional precision, 12.4 for more integer range – but 8.8 covers the vast majority of use cases. The game development chapters later in this book use 8.8 exclusively.

Theory and Practice

These algorithms are not isolated techniques. They form a system. Multiply feeds the rotation matrix. Rotation outputs coordinates needing perspective division. Division uses log tables. Projected vertices connect with lines drawn by the matrix method. All of it runs on fixed-point arithmetic, with sine values from the parabolic table.

Dark designed these as components of a single engine – the engine that powered *Illusion*. A wireframe cube spinning at full frame rate exercises every routine in this chapter:

1. **Read the rotation angle** from the sine table (parabolic approximation, ~20 T-states per lookup)
2. **Multiply** vertex coordinates by rotation factors (shift-and-add for accuracy, or square-table for speed – ~200 or ~60 T-states per multiply, 12 multiplies per vertex)
3. **Divide** by Z for perspective projection (log tables, ~60 T-states per division)
4. **Draw lines** between projected vertices (matrix Bresenham, ~48 T-states per pixel)

For a simple cube (8 vertices, 12 edges), the total per-frame cost is roughly:

- Rotation: 8 vertices x 12 multiplies x 200 T-states = 19,200 T-states
- Projection: 8 vertices x 1 divide x 60 T-states = 480 T-states
- Line drawing: 12 edges x ~40 pixels x 48 T-states = 23,040 T-states
- **Total: ~42,720 T-states** – comfortably within the ~70,000 T-state frame budget

Switch to the fast square-table multiply and rotation drops to 5,760 T-states. The vertices jitter slightly, but you now have headroom for more complex objects. Speed or accuracy – in a demo, you make that choice for every effect, every frame.

What Dark Got Right

Looking back at Spectrum Expert #01 from nearly thirty years' distance, what strikes you is not just the quality of the algorithms but the quality of the thinking. Dark presents each one, explains the trade-offs honestly, admits when his derivation has gaps, and trusts the reader to fill those gaps.

He was writing for Spectrum coders in Russia in the late 1990s – a community building some of the most impressive 8-bit demos in the world, on hardware the rest of the world had abandoned. These are the building blocks they used. When you write your first 3D engine for the Spectrum, these routines will make it possible.

In the next chapter, Dark and STS extend this mathematical foundation into a complete 3D system: the midpoint method for vertex interpolation, backface culling, and solid polygon rendering. The maths here is the foundation. Chapter 5 is the architecture built on top.

Random Numbers: When Tables Won't Do

Everything so far in this chapter is deterministic. Given the same inputs, the same multiply, the same sine lookup, the same line draw – you get the same output. That is exactly what you want for a spinning wireframe cube or a smooth plasma.

But sometimes you need chaos. Stars twinkling in a star field. Particles scattering from an explosion. Noise textures for terrain generation. A shuffled order for loading screens. In size-coding competitions (256 bytes or less), a good random number generator can produce surprisingly complex visual effects from almost no code.

The Z80 has no hardware random number generator. You must synthesize randomness from arithmetic, and the quality of that arithmetic matters more than you might think.

The R Register Trick

The Z80 has a built-in source of entropy that many coders reach for first: the R register. It increments automatically with each instruction fetch (every M1 cycle), cycling through 0-127. You can read it in 9 T-states:

```
z80 id:ch04_the_r_register_trick     ld   a, r              ; 9 T -- read refresh
counter
```

This is *not* a PRNG. The R register is fully deterministic – it advances by one per instruction, and its value at any point depends entirely on the code path taken since reset. In a demo with a fixed main loop, R produces the same sequence every time. But it is useful as a seed source: read R once at startup (when timing depends on how long the user waited before pressing a key) and feed that unpredictable value into a proper PRNG.

Some coders mix R into their generator at every call, adding genuine instruction-timing entropy. The Ion generator below uses exactly this trick.

Four Generators from the Community

In 2024, Gogin (of the Russian ZX scene) assembled a collection of Z80 PRNG routines and shared them for evaluation. Gogin tested them systematically, filling large bitmaps to reveal statistical patterns. The results are instructive – not all "random" routines are equally random.

Here are four generators from that collection, ordered from best to worst quality.

Patrik Rak (Raxoft)'s CMWC Generator (Best Quality)

This is a **Complement Multiply-With-Carry** generator by Patrik Rak (Raxoft), using the multiplier 253 and an 8-byte circular buffer. The mathematics behind CMWC are well-studied: George Marsaglia proved that certain multiplier/buffer combinations produce sequences with enormous periods. With multiplier 253 and buffer size 8, the theoretical period is (253^8 - 1) / 254 – approximately 2^66 values before repeating.

"'z80 id:ch04_four_generators_from_the ; Patrik Rak's CMWC PRNG ; Quality: Excellent – passes visual bitmap tests ; Size: ~30 bytes code + 8 bytes table ; Output: A = pseudo-random byte ; Period: ~2^66

patrik_rak_cmwc_rnd: ld hl, .table .smc_idx: ld bc, 0 ; 10 T – i (self-modifying) add hl, bc ; 11 T ld a, c ; 4 T inc a ; 4 T and 7 ; 7 T – wrap index to 0-7 ld (.smc_idx+1), a ; 13 T – store new index ld c, (hl) ; 7 T – y = q[i] ex de, hl ; 4 T ld h, c ; 4 T – t = 256 * y ld l, b ; 4 T sbc hl, bc ; 15 T – t = 255 * y sbc hl, bc ; 15 T – t = 254 * y sbc hl, bc ; 15 T – t = 253 * y .smc_car: ld c, 0 ; 7 T – carry (self-modifying) add hl, bc ; 11 T – t = 253 * y + c ld a, h ; 4 T ld (.smc_car+1), a ; 13 T – c = t / 256 ld a, l ; 4 T – x = t % 256 cpl ; 4 T – x = ~x (complement) ld (de), a ; 7 T – q[i] = x ret ; 10 T

.table: DB 82, 97, 120, 111, 102, 116, 20, 12

```
The algorithm multiplies the current buffer entry by 253, adds a carry value, stores the new c
byte circular buffer means the generator's state space is vast -- 8 bytes of buffer plus 1 byt
register generator can achieve.

Gogin's verdict: **best quality** in the collection. When filling a 256x192 bitmap, no visible

#### Ion Random (Second Best)

Originally from Ion Shell for the TI-83 calculator, adapted for Z80. This generator mixes the

```z80 id:ch04_four_generators_from_the_2
; Ion Random
; Quality: Good -- minor patterns visible only at extreme scale
; Size:    ~15 bytes
; Output:  A = pseudo-random byte
; Origin:  Ion Shell (TI-83), adapted for Z80

ion_rnd:
.smc_seed:
    ld   hl, 0              ; 10 T -- seed (self-modifying)
    ld   a, r               ; 9 T -- read refresh counter
    ld   d, a               ; 4 T
    ld   e, (hl)            ; 7 T
    add  hl, de             ; 11 T
    add  a, l               ; 4 T
    xor  h                  ; 4 T
    ld   (.smc_seed+1), hl  ; 16 T -- update seed
    ret                     ; 10 T
```

The R register injection means this generator produces different sequences depending on the calling context – how many instructions execute between calls affects R, which feeds back into the state. For a demo main loop with fixed timing, R advances predictably, but the nonlinear mixing (ADD + XOR) still produces good output. In a game where player input varies the call pattern, the R contribution adds genuine unpredictability.

Gogin's verdict: **second best**. Very compact, good quality for its size.
```

XORshift 16-bit (Mediocre)

A 16-bit XORshift generator – the Z80 adaptation of Marsaglia's well-known family:

"'z80 id:ch04_four_generators_from_the_3 ; 16-bit XORshift PRNG ; Quality: Mediocre – visible diagonal patterns in bitmap tests ; Size: ~25 bytes ; Output: A = pseudo-random byte (H or L) ; Period: 65535

xorshift_rnd: .smc_state: ld hl, 1 ; 10 T – state (self-modifying, must not be 0) ld a, h ; 4 T rra ; 4 T ld a, l ; 4 T rra ; 4 T xor h ; 4 T ld h, a ; 4 T ld a, l ; 4 T rra ; 4 T ld a, h ; 4 T rra ; 4 T xor l ; 4 T ld l, a ; 4 T xor h ; 4 T ld h, a ; 4 T ld (.smc_state+1), hl ; 16 T – update state ret ; 10 T

```
XORshift generators are fast and simple, but with only 16 bits of state the period is at most
rotation pattern creates visible diagonal streaks when the output is mapped to pixels. For a q

#### Patrik Rak's CMWC Variant (Mediocre)

A second CMWC variant by Patrik Rak (Raxoft), similar in principle to his version above but wi
byte-buffer version above is strictly superior.

### Elite's Tribonacci Approach

Worth a brief mention: the legendary *Elite* (1984) used a Tribonacci-like sequence for its pr
distributed sequences. The key insight was reproducibility -- given the same seed, the same ga
coding.

### Elite's Galaxy Generator: A Deeper Look

The Tribonacci approach deserves more detail because it illustrates a key principle: **a PRNG

David Braben and Ian Bell needed 8 galaxies of 256 star systems, each with a name, position, e
byte seed per galaxy and a deterministic generator that expanded each seed into the full star
register feedback loop -- each step rotates and XORs three 16-bit values:

```z80 id:ch04_elite_s_galaxy_generator_a
; Elite's galaxy generator (conceptual, 6502 origin):
;   seed = [s0, s1, s2]  (three 16-bit words)
;   twist: s0' = s1, s1' = s2, s2' = s0 + s1 + s2  (mod 65536)
;   repeat twist for each byte of star system data
```

On the Z80, the same principle works with three register pairs. The "twist" operation produces deterministic but well-distributed values. The crucial property: given the same seed, the same galaxy generates every time. Navigation between stars is just re-seeding and re-generating.

This idea – **small state, large apparent complexity** – drives demoscene size-coding too. A 256-byte intro that fills the screen with intricate patterns is doing exactly what Elite did: expanding a tiny seed into a large, complex output through a deterministic process.
```

Shaped Randomness

Sometimes you want numbers that are random but follow a specific distribution. A flat uniform PRNG gives every value equal probability, but real-world phenomena are rarely uniform: enemy spawn rates, particle speeds, terrain heights – all tend to cluster around preferred values.

Common tricks on the Z80:

- **Triangular distribution** – add two uniform random bytes and shift right. The sum clusters around the centre (128), producing "natural-looking" variation. Cost: two PRNG calls + ADD + SRL = ~20 extra T-states.

```
z80 id:ch04_shaped_randomness ; Triangular random: result clusters around 128
call patrik_rak_cmwc_rnd  ; A = uniform random     ld   b, a     call patrik_rak_cmwc_rnd
; A = another uniform random     add  a, b                ; sum (wraps at 256)
rra                       ; divide by 2 → triangular distribution
```

- **Rejection sampling** – generate a random number, reject values outside your desired range. For power-of-two ranges this is free (just AND with a mask). For arbitrary ranges, loop until the value fits.
- **Weighted tables** – store a 256-byte lookup table where each output value appears in proportion to its desired probability. Index with a uniform random byte. The table costs 256 bytes but the lookup is instant (7 T-states). Perfect when the distribution is complex and fixed.
- **PRNG as hash function** – feed structured data (coordinates, frame numbers) through the PRNG to get deterministic noise. This is how size-coded plasma and noise textures work: `random(x XOR y XOR frame)` gives a different-looking value per pixel per frame, but it is entirely reproducible.

Seeds and Reproducibility

In a demo, reproducibility is usually desirable: the effect should look the same every time it runs, because the coder choreographed the visuals to match the music. Seed the PRNG once with a fixed value and the sequence is deterministic.

In a game, unpredictability matters. Common seeding strategies:

- **FRAMES system variable ($5C78)** – the Spectrum ROM maintains a 3-byte frame counter at address $5C78 that increments every 1/50th of a second from power-on. Reading it gives a time-dependent seed that varies with how long the machine has been running. Art-top (Artem Topchiy) recommends using it to initialise Patrik Rak's CMWC table:

```
z80 id:ch04_seeds_and_reproducibility ; Seed Patrik Rak CMWC from FRAMES system
variable     ld   hl, $5C78            ; FRAMES (3 bytes, increments at 50
Hz)     ld   a, (hl)              ; low byte -- most variable     ld   de,
patrik_rak_cmwc_rnd.table     ld   b, 8 .seed_loop:     xor  (hl)                ;
mix with FRAMES     ld   (de), a              ; write to table     inc  de
rlca                      ; rotate for variety     add  a, b                ;
add loop counter     djnz .seed_loop
```

- **Read R at a user-input moment** – the exact instruction count between reset and the player pressing a key varies each run. `LD A,R` at that moment captures timing entropy.

- **Frame counter accumulation** – XOR the R register into an accumulator every frame during the title screen; use the accumulated value as seed when the game starts.
- **Combine multiple sources** – XOR together R, the low byte of FRAMES, and a byte from the floating bus (on 48K Spectrums, reading certain ports returns whatever the ULA is currently fetching from RAM – a source of positional entropy).

For demos, simply initialise the generator's state to a known value and leave it. The compilable example (`examples/prng.a80`) shows all four generators with fixed seeds.

Comparison Table

Algorithm	Size (bytes)	Speed (T-states)	Quality	Period	Notes
Patrik Rak CMWC	~30 + 8 table	~170	Excellent	~2^66	Best overall; 8-byte buffer
Ion Random	~15	~75	Good	Depends on R	Compact; mixes R register
XORshift 16	~25	~90	Mediocre	65,535	Visible diagonal patterns
Patrik Rak CMWC (alt)	~35 + 10 table	~180	Mediocre	~2^66	Patterns visible at scale
LD A,R alone	2	9	Poor	128	NOT a PRNG; use as seed only

For most demoscene work, **Patrik Rak's CMWC** is the clear winner: excellent quality, reasonable size, and a period so long it will never repeat during a demo. If code size is critical (size-coding, 256-byte intros), **Ion Random** packs remarkable quality into 15 bytes. XORshift is a fallback when you need something quick and do not care about visual quality.

> **Credits:** PRNG collection, quality assessment, and bitmap testing by **Gogin**. Patrik Rak's CMWC generator is based on George Marsaglia's Complementary Multiply-With-Carry theory. Ion Random originates from **Ion Shell** for the TI-83 calculator.

All cycle counts in this chapter are for Pentagon timing (no wait states). On a standard 48K Spectrum or Scorpion with contended memory, expect higher counts for code executing in the lower 32K of RAM. See Appendix A for the complete timing reference.

Chapter 5: 3D on 3.5 MHz

> *"Calculate only what you must. Derive the rest."* — Dark & STS, Spectrum Expert #02 (1998)

The previous chapter gave you the building blocks: multiplication, division, sine tables, line drawing. Now we put them together. The goal is a spinning three-dimensional solid object on a ZX Spectrum — filled polygons, backface culling, correct depth ordering — at a usable frame rate.

This is where you hit the wall.

The Problem: Twelve Multiplications Per Vertex

Rotating a point in three-dimensional space around all three axes requires a sequence of trigonometric multiplications. If you rotate sequentially — first around Z, then Y, then X — each axis involves four multiplications and two additions to transform two coordinates. Three axes, four multiplications each: twelve multiplications per vertex.

Take the shift-and-add multiply from Chapter 4, which costs roughly 200 T-states. Twelve of those give you 2,400 T-states to rotate a single vertex. A simple cube has 8 vertices: 19,200 T-states just for rotation. Chapter 4 showed that this fits in the frame budget — barely.

Now try something more interesting. A sphere approximated by 20 vertices and 36 faces:

```
20 vertices x 2,400 T-states = 48,000 T-states
```

That is 67% of the Pentagon's 71,680 T-state frame budget, consumed before you have drawn a single pixel. You still need perspective projection, backface culling, polygon sorting, and the actual fill. There is no room. The object cannot be more complex than a cube unless you find a fundamentally cheaper way to compute vertex positions.

Dark and STS found one.

The Midpoint Method

The insight is geometric. Not every vertex in an object carries independent information. Many vertices sit at structurally predictable positions — midpoints of edges,

centers of faces, reflections of other vertices. If you can express those relationships explicitly, you can replace expensive multiplications with cheap averages.

The Cube as Foundation

Consider a cube centered at the origin. It has 8 vertices, but they are not 8 independent points. They are 4 pairs of diametrically opposite vertices. If you know one vertex of a pair, the other is its negation through the center:

```
v0 = ( x,  y,  z)    →    v7 = (-x, -y, -z)
v1 = ( x,  y, -z)    →    v6 = (-x, -y,  z)
v2 = ( x, -y,  z)    →    v5 = (-x,  y, -z)
v3 = ( x, -y, -z)    →    v4 = (-x,  y,  z)
```

Rotate 4 vertices using the full 12-multiplication procedure. Negate them to get the other 4. Negation on the Z80 is `NEG` — 8 T-states for one coordinate, 24 T-states for all three. Compare that to 2,400 T-states for a full rotation. You have cut the vertex computation nearly in half.

But the midpoint method goes much further than mirroring.

Deriving Vertices by Averaging

The key operation is the average: given two already-computed points, their midpoint is simply the mean of their coordinates.

```
v_new = (v_a + v_b) / 2
```

On the Z80, this is an addition and a shift:

“‘z80 id:ch05_deriving_vertices_by_2 ; Average two signed 8-bit coordinates ; A = first coordinate, B = second coordinate ; Result in A = (A + B) / 2

```
add  a, b           ;  4 T-states
sra  a              ;  8 T-states
                    ; ----
                    ; 12 T-states total


`SRA` (Shift Right Arithmetic) preserves the sign bit, so this works correctly for negative co
states per derived vertex. Compare that to 2,400 T-states for a full rotation.

The ratio: averaging is **66 times cheaper** than rotation.

This means you can build complex objects from a small set of "basis" vertices that you rotate

### Building Complex Objects

Suppose you want a 20-vertex object. With the midpoint method:

1. Rotate 4 basis vertices fully: 4 x 2,400 = 9,600 T-states
2. Mirror 4 vertices by negation: 4 x 24 = 96 T-states
3. Derive 12 vertices by averaging: 12 x 36 = 432 T-states
4. **Total: 10,128 T-states**
```

```
Without the midpoint method, the same 20 vertices would cost 48,000 T-states. You have saved 3
states --- more than half the frame budget freed up for projection, culling, and rendering.

The constraint is topological: you can only derive a vertex by averaging if it genuinely lies

Dark and STS give examples of the derivation chains:

```text
v8  = (v4 + v5) / 2
v9  = (v3 + v7) / 2
v10 = (v2 + v6) / 2
v11 = (v8 + v9) / 2     ; derived from two already-derived vertices
```

Notice that v11 is derived from v8 and v9, which are themselves derived. Chains can go several levels deep. Each level adds only 36 T-states per vertex, so the cost stays negligible regardless of depth.

The Virtual Processor

Here is where Dark does something that feels anachronistic for 1998. Rather than hardcoding the derivation chains for each specific object, he designs a tiny interpreter — a virtual processor — that executes "programs" describing how to compute vertices.

Architecture

The virtual processor has:

- **One register** (a working register holding one 3D point — three bytes: x, y, z)
- **64 cells of RAM** (each cell holds one 3D point — 192 bytes total)
- **4 instructions**

Opcode	Bits	Name	Operation
00	`00nnnnnn`	**Load**	register <– cell[n]
01	`01nnnnnn`	**Store**	cell[n] <– register
10	`10nnnnnn`	**Average**	register <– (register + cell[n]) / 2
11	`11------`	**End**	halt execution

Each instruction is encoded in a single byte: 2 bits for the opcode, 6 bits for the cell number (0–63). The entire instruction set fits in 256 possible values.

Execution

The interpreter loop is compact:

"'z80 id:ch05_execution ; Virtual processor main loop ; IX points to the program (sequence of 1-byte instructions) ; Point RAM at a fixed address, 3 bytes per cell
```

vp_loop: ld a, (ix+0) ; fetch instruction inc ix ld b, a ; save full instruction and %11000000 ; extract opcode (top 2 bits)

```
cp   %11000000        ; END?
ret  z                ; yes — halt

ld   a, b
and  %00111111        ; extract cell number (bottom 6 bits)
; ... compute cell address from cell number ...
; ... dispatch based on opcode ...

jr   vp_loop


The **Load** instruction copies a cell's x, y, z values into the working register. **Store** c

### Writing Programs

A vertex derivation chain becomes a simple sequence of bytes. Dark uses a compact notation in

```z80 id:ch05_writing_programs
; Example: derive v8 = (v4 + v5) / 2, then store it
; Cell 4 = v4, Cell 5 = v5, Cell 8 = destination

    DB  4           ; LOAD cell[4]     (opcode 00, cell 4)
    DB  128+5       ; AVG  cell[5]     (opcode 10, cell 5 = %10000101)
    DB  64+8        ; STORE cell[8]    (opcode 01, cell 8 = %01001000)
```

The notation `128+5` encodes `%10000101` — opcode 10 (Average) with cell number 5. `64+8` encodes `%01001000` — opcode 01 (Store) with cell number 8. Raw numbers, packed into data bytes, forming a tiny domain-specific program.

A complete object description might look like:

“‘z80 id:ch05_writing_programs_2 ; Midpoint program for a 12-vertex object ; Cells 0-3: basis vertices (rotated by main code) ; Cells 4-7: mirrored vertices (negated by main code) ; Cells 8-11: derived via midpoint averaging

midpoint_program: DB 0 ; LOAD v0 DB 128+1 ; AVG v1 -> register = (v0+v1)/2 DB 64+8 ; STORE v8

```
DB  2               ; LOAD v2
DB  128+3           ; AVG  v3           -> register = (v2+v3)/2
DB  64+9            ; STORE v9

DB  4               ; LOAD v4
DB  128+5           ; AVG  v5           -> register = (v4+v5)/2
DB  64+10           ; STORE v10

DB  6               ; LOAD v6
DB  128+7           ; AVG  v7           -> register = (v6+v7)/2
DB  64+11           ; STORE v11

DB  192             ; END               (%11000000)
```
```

```
Thirteen bytes describe the computation of four derived vertices. The virtual processor execut
states (each instruction takes approximately 25--35 T-states depending on type). Four fully ro
states. The savings are enormous.

### Why a Virtual Processor?

You might ask: why not just write the averaging code directly in Z80 assembly? Inline the addi

The answer is flexibility. The virtual processor separates the *description* of an object's to

This is, in essence, a domain-specific bytecode interpreter --- a pattern that modern programm
states on vertex computation. The architecture is clean.

---

## Rotation

With the midpoint method handling most vertices, you still need to rotate the basis vertices p

### Z-Axis Rotation

Rotation around Z affects only X and Y:

```text
X' = X * cos(Az) + Y * sin(Az)
Y' = -X * sin(Az) + Y * cos(Az)
```

In Z80 assembly, using the 8x8 signed multiply and 256-entry sine/cosine tables:

“‘z80 id:ch05_z_axis_rotation_2 ; Rotate point around Z axis ; Input: (px), (py) = coordinates; (angle_z) = rotation angle ; Output: (px), (py) updated ; Uses: cos_table, sin_table (page-aligned, signed 8-bit)

rotate_z: ld a, (angle_z) ld l, a ld h, cos_table » 8 ld d, (hl) ; D = cos(Az) ld h, sin_table » 8 ld e, (hl) ; E = sin(Az)

```
; X' = X*cos(Az) + Y*sin(Az)
ld   a, (px)
ld   b, a
ld   c, d               ; B=X, C=cos
call mul_signed          ; HL = X * cos(Az)
push hl

ld   a, (py)
ld   b, a
ld   c, e               ; B=Y, C=sin
call mul_signed          ; HL = Y * sin(Az)
pop  de
add  hl, de             ; HL = X*cos + Y*sin
ld   a, h               ; take high byte as new X'
ld   (px), a
```
```

```
; Y' = -X*sin(Az) + Y*cos(Az)
ld   a, (px_original)   ; need the original X, not the updated one
neg
ld   b, a
ld   c, e               ; B=-X, C=sin
call mul_signed          ; HL = -X * sin(Az)
push hl

ld   a, (py)
ld   b, a
ld   c, d               ; B=Y, C=cos
call mul_signed          ; HL = Y * cos(Az)
pop  de
add  hl, de             ; HL = -X*sin + Y*cos
ld   a, h
ld   (py), a

ret


The same pattern repeats for Y-axis rotation (affecting X and Z) and X-axis rotation (affectin

Note the detail about preserving the original X value. The second formula uses the pre-
rotation X, not the X' you just computed. A common bug is to use the already-
updated coordinate, which produces a skewed rotation. Dark addresses this explicitly in the ar

### Cost Per Basis Vertex

Each axis rotation requires 4 multiplications and 2 additions. At 200 T-states per multiply an
states per 16-bit add:

```text
Per axis:  4 x 200 + 2 x 11 = 822 T-states
Three axes: 3 x 822 = 2,466 T-states per vertex
```

With 4 basis vertices: roughly 9,864 T-states for rotation. Add the midpoint program execution and you have the full vertex computation for an arbitrarily complex object at a fraction of the naive cost.

Projection

Once all vertices are rotated in 3D space, you need to project them onto the 2D screen.

Parallel Projection

The simplest approach: ignore the Z coordinate entirely. Just use X and Y as screen coordinates (with appropriate offset to center the object on screen).

“‘z80 id:ch05_parallel_projection ; Parallel projection: screen coords = rotated X, Y + offset ld a, (px) add a, 128 ; center horizontally (128 = half of 256) ld (screen_x),
```

a

```
ld   a, (py)
add  a, 96              ; center vertically (96 = half of 192)
ld   (screen_y), a


Cost: essentially zero. The result looks flat --- objects do not appear to recede into the dis

### Perspective Projection

Perspective makes near objects larger and far objects smaller, producing the depth cue that ma

```text
Xscreen = (X * Scale) / (Z + Zdistance) + Xoffset
Yscreen = (Y * Scale) / (Z + Zdistance) + Yoffset
```

`Scale` controls the field of view. `Zdistance` is the distance from the camera to the projection plane — it prevents division by zero when Z approaches the camera and controls how aggressively objects scale with depth. `Xoffset` and `Yoffset` center the projection on screen.

The expensive operation here is the division. One divide per coordinate, two coordinates per vertex. Using the logarithmic division from Chapter 4 (~60 T-states per divide), the cost is modest:

“‘z80 id:ch05_perspective_projection_2 ; Perspective projection for one vertex ; Input: (px), (py), (pz) = rotated 3D coordinates ; Output: (screen_x), (screen_y)

perspective: ; Compute denominator: Z + Zdistance ld a, (pz) add a, ZDISTANCE ; Z + viewing distance ld c, a ; C = denominator

```
; Xscreen = (X * Scale) / (Z + Zdist) + Xoffset
ld   a, (px)
ld   b, SCALE
call mul_signed          ; HL = X * Scale
ld   a, h               ; take high byte as numerator
call log_divide          ; A = A / C (using log tables)
add  a, XOFFSET
ld   (screen_x), a

; Yscreen = (Y * Scale) / (Z + Zdist) + Yoffset
ld   a, (py)
ld   b, SCALE
call mul_signed          ; HL = Y * Scale
ld   a, h
call log_divide          ; A = A / C
add  a, YOFFSET
ld   (screen_y), a

ret


Each vertex costs two multiplications (400 T-states) and two log divides (120 T-
states), plus overhead --- roughly 600 T-states per vertex. For 20 vertices: 12,000 T-
```
```

```
states. Combined with the midpoint rotation, we are at roughly 22,000 T-states for all vertex

---

## Solid Polygons

A wireframe object is a collection of edges. A solid object is a collection of filled polygons

### Backface Culling

A closed 3D object has faces that point toward the viewer and faces that point away. The back-
facing polygons are hidden and need not be drawn. Skipping them saves both rendering time and

The test is geometric. For each face, compute the Z-component of the surface normal using the

```text
Given three vertices of a face: v0, v1, v2

Edge vectors:
    V = v1 - v0 = (Vx, Vy)    (in screen coordinates)
    W = v2 - v0 = (Wx, Wy)

Z-component of normal = Vx * Wy - Vy * Wx
```

If the result is positive, the face is oriented toward the viewer and should be drawn. If negative, the face points away — cull it. If zero, the face is edge-on and invisible.

“‘z80 id:ch05_backface_culling_2 ; Backface culling test for one face ; Input: three projected vertices (x0,y0), (x1,y1), (x2,y2) ; Output: carry flag set if face is back-facing (should be culled)

backface_test: ; V = v1 - v0 ld a, (x1) sub (ix+x0) ld d, a ; D = Vx = x1 - x0

```
ld   a, (y1)
sub  (ix+y0)
ld   e, a               ; E = Vy = y1 - y0

; W = v2 - v0
ld   a, (x2)
sub  (ix+x0)
ld   b, a               ; B = Wx = x2 - x0

ld   a, (y2)
sub  (ix+y0)
ld   c, a               ; C = Wy = y2 - y0

; Normal Z = Vx * Wy - Vy * Wx
ld   a, d
call mul_signed_c        ; HL = Vx * Wy (D * C)
push hl

ld   a, e
ld   c, b
```
```

```
call mul_signed_c        ; HL = Vy * Wx (E * B)

pop  de
ex   de, hl
or   a
sbc  hl, de              ; HL = Vx*Wy - Vy*Wx

bit  7, h                ; check sign
ret                      ; carry/sign indicates facing


Two multiplications and a subtraction per face. At 400 T-states for the multiplies plus overhe
states per face. For a 12-face object, that is 6,000 T-states --- and for each culled face, yo

On a typical rotating solid, roughly half the faces are back-facing at any time. Culling them

### Z-Sorting

For a convex object (a cube, a tetrahedron), backface culling alone produces correct results:
object scenes, you need to draw faces in back-to-front order so that nearer faces overwrite fa

Dark and STS compute a depth value for each visible face (typically the average Z of its verti
12 faces takes negligible time compared to filling them.

```z80 id:ch05_z_sorting
; Simplified depth sort: compute average Z for each visible face,
; sort face indices by descending Z (farthest first)

sort_faces:
    ; For each visible face:
    ;   average_z = (z[v0] + z[v1] + z[v2] + z[v3]) / 4
    ;   store (average_z, face_index) in sort buffer
    ; Then insertion-sort the buffer by average_z
    ; ...
```

Convex Polygon Filling

Once you know which faces to draw and in what order, you need to fill them. A convex polygon (all interior angles less than 180 degrees) can be filled with a simple scanline approach:

1. Find the topmost and bottommost vertices.
2. Walk down the left edge and the right edge simultaneously, one scanline at a time.
3. For each scanline, draw a horizontal line from the left edge to the right edge.

The edge-walking uses Bresenham-style incremental stepping — no division needed per scanline, just additions and conditional increments. The horizontal fill itself is a tight loop of byte writes:

“‘z80 id:ch05_convex_polygon_filling ; Fill one scan line from x_left to x_right at screen row Y ; Screen address already computed in HL

fill_scanline: ld a, (x_right) sub (ix+x_left) ret c ; nothing to fill if right < left ret z
```

ld b, a ; B = pixel count

```
; For byte-aligned fills: write whole bytes
ld   a, $FF             ; solid fill
```

.fill_loop: ld (hl), a inc l ; next byte (within same screen line) djnz .fill_loop ret

```
This is simplified --- real polygon fillers must handle partial bytes at the left and right ed

---

## Putting It All Together

The complete frame loop for a spinning 3D solid object follows this sequence:

<!-- figure: ch05_3d_pipeline -->
![3D rendering pipeline: model, rotation, projection, screen](illustrations/output/ch05_3d_pip

```text
1. Update rotation angles (Az, Ay, Ax)
2. For each basis vertex:
     Rotate through Z, Y, X axes         [~2,400 T per vertex]
3. Negate basis vertices to get mirrors   [~24 T per vertex]
4. Run midpoint program to derive rest    [~36 T per derived vertex]
5. Project all vertices (perspective)     [~600 T per vertex]
6. For each face:
     Backface test                         [~500 T per face]
     If visible: compute average Z
7. Sort visible faces by Z               [~200 T for small lists]
8. For each visible face (back to front):
     Fill polygon                          [varies with area]
9. Wait for next frame (HALT)
```

For a 20-vertex, 18-face object with 4 basis vertices, the per-frame budget breaks down as:

Stage	Vertices/Faces	Cost per	Total
Rotation (basis)	4	2,466	9,864
Negation (mirrors)	4	24	96
Midpoint derivation	12	36	432
Projection	20	600	12,000
Backface test	18	500	9,000
Z-sort	~9 visible	-	~200
Polygon fill	~9 visible	~1,500 avg	~13,500
Total			**~45,092**

Forty-five thousand T-states out of 71,680 available. Tight, but workable — you have 26,000 T-states remaining for screen clearing, angle updates, and the line drawing or edge-highlighting that makes the object look crisp. And this is for a 20-vertex object, far more complex than anything you could afford with naive rotation.
```

The Shape of Objects

The midpoint method influences how you think about 3D models. You do not design a mesh and then optimize it — you start from the topology that the method requires.

A good midpoint object begins with a small basis. Four fully rotated points define a tetrahedron-like skeleton. Negation doubles that to eight. Midpoint averaging fills in the rest. The art is choosing basis vertices that produce useful derived points.

Consider building a 14-vertex object from scratch:

```
Basis:    v0, v1, v2, v3         (4 fully rotated)
Mirrors:  v4, v5, v6, v7         (4 negated)
Derived:
  v8  = (v0 + v1) / 2           edge midpoint
  v9  = (v2 + v3) / 2           edge midpoint
  v10 = (v4 + v5) / 2           edge midpoint on mirrored side
  v11 = (v6 + v7) / 2           edge midpoint on mirrored side
  v12 = (v0 + v2) / 2           cross-edge midpoint
  v13 = (v8 + v10) / 2          second-level derivation
```

The virtual processor program for this is 19 bytes:

```
z80 id:ch05_the_shape_of_objects_2 object_14v_program:     DB  0, 128+1, 64+8
; v8  = avg(v0, v1)     DB  2, 128+3, 64+9      ; v9  = avg(v2, v3)     DB  4,
128+5, 64+10     ; v10 = avg(v4, v5)     DB  6, 128+7, 64+11     ; v11 = avg(v6,
v7)     DB  0, 128+2, 64+12     ; v12 = avg(v0, v2)     DB  8, 128+10, 64+13
; v13 = avg(v8, v10)     DB  192                  ; END
```

Nineteen bytes of data replace what would otherwise be 14,400 T-states of rotation multiplications for the 6 derived vertices.

You can push this further. A second level of averaging (deriving from already-derived points) costs nothing extra per instruction — the virtual processor does not care whether a cell holds a rotated or a derived point. Dark and STS describe chains three or four levels deep, building objects with 30 or more vertices from just 3–4 basis points.

The constraint is precision. Each averaging step introduces a rounding error of up to 0.5 units (from the integer shift). After three levels of derivation, the cumulative error can reach 1.5 units — noticeable on an object that spans 60 pixels, invisible on one that spans 120. Design your objects large enough that the rounding is below the screen resolution.

Practical: A Spinning Solid Object

Here is the outline for building a complete spinning 3D solid using everything in this chapter.

Step 1: Define the Object

Start with basis vertices. A truncated octahedron works well with the midpoint method:

```
z80 id:ch05_step_1_define_the_object ; 4 basis vertices in signed 8-bit coordinates
basis_vertices:     DB   30,   0,  30     ; v0 (x, y, z)     DB    0,  30,  30
; v1     DB   30,  30,   0     ; v2     DB    0,   0,   0     ; v3 (at origin
for center reference)
```

Step 2: Write the Midpoint Program

```
z80 id:ch05_step_2_write_the_midpoint midpoint_prog:     ; Mirrors: cells 4-7
are pre-negated by the main loop     ; Derive additional vertices:     DB  0,
128+1, 64+8        ; v8  = avg(v0, v1)     DB  2, 128+3, 64+9        ; v9  =
avg(v2, v3)     DB  0, 128+2, 64+10       ; v10 = avg(v0, v2)     DB  1, 128+3,
64+11       ; v11 = avg(v1, v3)     DB  192                    ; END
```

Step 3: Define Faces

```
z80 id:ch05_step_3_define_faces ; Face table: each face is a list of vertex
indices + attribute byte ; Vertex order must be consistent (clockwise when
front-facing) face_table:     DB  4, 0, 1, 8, 10        ; face 0: quad (4 vertices)
DB  4, 2, 3, 9, 11        ; face 1: quad     ; ... remaining faces ...     DB  0
; end marker
```

Step 4: The Frame Loop

“‘z80 id:ch05_step_4_the_frame_loop main_loop: halt ; wait for vsync (IM1)

```
; Clear the screen area (or use double buffering)
call clear_viewport

; Update angles
ld   hl, angle_z
inc  (hl)
ld   hl, angle_x
ld   a, (hl)
add  a, 2
ld   (hl), a

; Rotate basis vertices
ld   b, 4                  ; 4 basis vertices
ld   ix, basis_vertices
ld   iy, point_ram         ; cell 0 onwards
```

.rotate_basis: push bc call rotate_xyz ; rotate point at (IX) by current angles ; store result at (IY) ld bc, 3 add ix, bc add iy, bc pop bc djnz .rotate_basis

```
; Negate for mirrors (cells 4-7 = negation of cells 0-3)
call negate_basis

; Run midpoint program
ld   ix, midpoint_prog
```

```
call virtual_processor

; Project all vertices
call project_all

; Backface cull and sort
call cull_and_sort

; Draw visible faces
call draw_faces

jr   main_loop
```

This is the skeleton. Each `call` hides a procedure built from the techniques in this chapter

---

Historical Context: From Magazine to Demo

Dark and STS published the midpoint method in Spectrum Expert #02 in 1998. They were young cod

But Spectrum Expert was not an academic exercise. Dark was from X-Trade, the same group that p
winning demo code. The sine tables from Chapter 4 drove the rotozoomer. The line drawer render

The virtual processor approach is particularly striking in retrospect. In 1998, the dominant p

The connection to *Illusion* runs deeper than shared authorship. When Introspec disassembled *
and-add multiplier, the parabolic sine table, the log-table divider. The midpoint method and t
winning demo.

In the next chapter, we will see one of *Illusion*'s most spectacular effects up close: the te
mapped sphere. It uses the same sine tables and the same fixed-point arithmetic from Chapter 4
modifying code techniques from Chapter 3 and a completely different approach to the rendering
states.

Summary

- Rotating 3D objects naively requires 12 multiplications per vertex --- too expensive for com
- The **midpoint method** rotates only a few basis vertices fully, then derives the rest throu
states per vertex versus ~2,400 for full rotation --- 66 times cheaper.
- A **virtual processor** with 4 instructions (Load, Store, Average, End) executes compact "pr
- **Rotation** uses sequential Z/Y/X axis transforms with sine/cosine lookup tables from Chapt
- **Perspective projection** uses the logarithmic division from Chapter 4 for the Z-
divide.
- **Backface culling** via cross-product normal test eliminates invisible faces at ~500 T-
states each.
- **Z-sorting** with the painter's algorithm handles overlapping faces for non-
convex objects.
- A 20-vertex solid object can be rendered in ~45,000 T-states per frame --- tight but feasibl

```
state budget.
- These techniques were published in Spectrum Expert #02 (1998) by the same team that built *I

---

*All cycle counts in this chapter are for Pentagon timing (no wait states). On a standard 48K

> **Sources:** Dark & STS, "Programming: 3D Graphics" (Spectrum Expert #01, 1997); Dark & STS,


\newpage

# Chapter 6: The Sphere --- Texture Mapping on 3.5 MHz

> *"Coder effects are always about evolving a computation scheme."*
> --- Introspec, 2017

---

It is 1996, and a demo called *Illusion* takes first place at ENLiGHT'96 in St. Petersburg. Th
processor. Just a Z80, 48 kilobytes of contiguous RAM, and whatever a twenty-
year-old coder named Dark could squeeze out of them.

Twenty years later, in March 2017, Introspec sits down with a copy of the binary and a disasse
states, maps memory addresses to data structures, and publishes his findings on Hype. What fol
time rendering on constrained hardware.

This chapter follows Introspec's analysis. We will look over his shoulder as he traces the cod

---

## The Problem: A Round Object on a Square Screen

A sphere on screen is not a sphere. It is a circle filled with a distorted image. The distorti
dimensional surface.

The source image in Illusion is stored as a monochrome bitmap --- one byte per pixel, where ea

The task, then, is this: read pixels from the source image, sample them according to a spheric

## The Insight: Code That Writes Code

The first question any Z80 programmer asks is: what does the inner loop look like? On a machin
states and you have roughly 70,000 T-states per frame, the inner loop *is* the program. Everyt

Dark's solution is to not have a fixed inner loop at all.

Instead, the rendering code is *generated at runtime*. For each horizontal line of the sphere,

This is a technique that appears throughout the demoscene: self-generating code, sometimes cal
```

```
## Inside the Disassembly

Introspec traced the rendering engine to a block of generated code and a set of lookup tables

At the equator, the skip distances are roughly uniform --- the source image maps onto the sphe

The generated inner loop has a repeating structure. For each screen byte (eight packed pixels)

```z80 id:ch06_inside_the_disassembly
; --- Accumulating 8 source pixels into one screen byte ---
; HL points into the source image (one byte per pixel)
; A is the accumulator, building the screen byte bit by bit

    add  a,a          ; shift accumulator left (make room for next pixel)
    add  a,(hl)       ; add source pixel (0 or 1) into lowest bit
    inc  l            ; advance to next source pixel
    ; ... possibly more INC L instructions here,
    ; depending on how many pixels to skip

    add  a,a          ; shift again
    add  a,(hl)       ; sample next pixel
    inc  l
    inc  l            ; skip one extra pixel (sphere curvature)

    add  a,a
    add  a,(hl)
    inc  l

    ; ... six more times, for 8 pixels total ...
```

The key detail: between each `add a,(hl)`, the number of `inc l` instructions varies. One pixel position might need a single `inc l` (sample adjacent pixels). Another might need three or four (skip over compressed regions of the projection). The lookup tables at `$6944` encode exactly how many `inc l` instructions to insert at each position.

Let us look more carefully at what happens with a single pixel:

```
z80 id:ch06_inside_the_disassembly_2     add  a,a          ;  4 T-states  (shift
A left by 1)     add  a,(hl)       ;  7 T-states  (add source pixel into bit 0)
inc  l            ;  4 T-states  (advance source pointer)
```

That is the minimum cost: 15 T-states to shift the accumulator and sample one pixel, plus 4 T-states for each source byte skipped. After eight such sequences, the accumulator holds a complete screen byte.

Notice that the source pointer is advanced using `inc l` rather than `inc hl`. This is deliberate. `INC HL` takes 6 T-states; `INC L` takes 4. By constraining the source data to sit within a single 256-byte page (so that only the low byte of the address changes), Dark saves 2 T-states per advance. When you are doing this thousands of times per frame, those 2 T-states add up.

There is a subtlety here that is easy to miss. The source image is stored as one byte per pixel, and `INC L` wraps around within a 256-byte page. This means each
```

scanline of source data must fit within 256 bytes, and the source buffer must be page-aligned. The constraint shapes the entire memory layout of the demo.

Counting T-States

Introspec calculated the cost per output byte as:

101 + 32x T-states

where *x* is the average number of extra `INC L` instructions per pixel beyond the mandatory one. Let us verify this.

The fixed cost per pixel is:

Instruction	T-states
`add a,a`	4
`add a,(hl)`	7
`inc l`	4
Subtotal	**15**

For 8 pixels, the fixed cost is 8 x 15 = 120 T-states. But there is additional overhead per byte: the code must write the completed byte to screen memory and set up for the next. Let us assume the output sequence looks something like:

```
z80 id:ch06_counting_t_states     ld   (de),a       ;  7 T-states  (write screen
byte)     inc  e            ;  4 T-states  (advance screen pointer)
```

There may also be setup for the accumulator (an `xor a` or similar) at the start of each byte. Taking Introspec's measured figure of 101 T-states as the fixed base cost per byte, the overhead beyond the raw pixel sampling accounts for roughly 101 - 120 = ... which means the base figure already includes the output instructions and some of the pixel work is interleaved differently than the naive count suggests.

The cleaner way to read the formula: 101 T-states of fixed overhead (output, pointer management, any per-byte setup), plus 32 T-states per extra skip. The "32" comes from 8 pixels times 4 T-states per extra `INC L`, which gives us x as the average number of extra skips per pixel position in that byte. When the sphere is near the equator, x is small — the projection is close to uniform. Near the poles, x is large, and the rendering slows down. But the poles also have fewer bytes to draw (the sphere is narrower there), so the total workload roughly balances.

Is this fast enough? The Spectrum's frame is approximately 70,000 T-states (more on Pentagon: 71,680). A 56-pixel-diameter sphere occupies roughly 7 bytes across at its widest. Over the full height, perhaps 200–250 bytes need to be rendered. At 101 T-states per byte (equatorial, x near zero), that is roughly 25,000 T-states — comfortably within a single frame budget, with room left for screen clearing, table lookups, and all the other housekeeping. Even near the poles where x might average 2–3, the cost per byte rises to 165–197 T-states, but fewer bytes need drawing. The arithmetic works out. It fits.

The Code Generation Pass

Before the inner loop runs, a *code generation pass* constructs it. This pass reads the lookup tables at `$6944`, which encode the sphere geometry for the current rotation angle, and emits Z80 instructions into a buffer:

1. For each scanline of the sphere, read the skip distances from the table.
2. Emit `add a,a` followed by `add a,(hl)` for each pixel.
3. Emit the appropriate number of `inc l` instructions based on the skip distance.
4. After every 8 pixels, emit the output instruction to write the accumulated byte to screen memory.
5. At the end of each scanline, emit a return or jump to the next line's handler.

The generated code block is then called directly. The CPU executes the instructions as if they were a normal subroutine, but they were written moments ago by the code generator. This is self-modifying code in the most literal sense — the program generates the program that draws the screen.

The code generation pass itself is not free, but it runs once per frame (or once per rotation step), while the generated inner loop runs hundreds of times. The amortized cost is negligible.

What Dark Knew: Spectrum Expert and the Building Blocks

There is a detail in this story that transforms it from a technical curiosity into a narrative arc. Dark — the coder behind Illusion's sphere effect — is the same Dark who wrote the *Programming Algorithms* articles in Spectrum Expert #01, published in 1997.

Those articles cover multiplication (shift-and-add vs. square-table lookup), division (restoring and logarithmic), sine table generation via parabolic approximation, and Bresenham line drawing with optimized 8x8 matrix blocks. They are tutorial material, written for the ZX Spectrum programming community, explaining fundamental techniques that any demo coder would need.

And they are, quite precisely, the building blocks used in Illusion.

The sphere requires: trigonometric lookup tables for computing the projection (sine/cosine, the parabolic approximation from Dark's article). Fixed-point multiplication for scaling. Careful memory layout for speed (the same cycle-counting discipline that Dark teaches throughout the articles). The skip-table approach to encoding sphere geometry is a direct application of the kind of precomputation-driven thinking that Dark advocates.

Dark wrote the demo first — Illusion won at ENLiGHT'96. Then, in 1997–98, he published the textbook that explained every technique he had used. Twenty years later, Introspec reverse-engineered the demo and found exactly the algorithms Dark had documented. We have both sides of the story: the practitioner explaining his methods after the fact, and the analyst confirming that those methods are precisely what the finished product contains.

The Hype Debate: Inner Loops vs. Mathematics

Introspec's 2017 article on Hype sparked a long comment thread. Among the most substantive exchanges was a debate between kotsoft and Introspec about where the real work of an effect like this lies.

kotsoft argued that the mathematical approach to the projection — how you compute which source pixel maps to which screen position — is the critical design decision. Get the projection wrong, or use a naive algorithm, and no amount of inner-loop optimization will save you. The mathematical model determines whether the effect is even *feasible* on the hardware.

Introspec countered that the inner loop is where the T-states are actually spent. You can have a beautiful mathematical model, but if the rendering code costs 200 T-states per byte instead of 100, you have halved your frame rate. The mathematical approach defines *what* to compute; the inner loop determines *whether you can compute it in time*.

Both are right, and the tension between them illuminates something fundamental about demoscene coding. A demo effect is not pure mathematics and not pure engineering. It is the intersection: an elegant computation scheme (the sphere projection encoded as skip tables) married to an efficient execution strategy (generated unrolled loops with `INC L` advances). Neither alone is sufficient.

Introspec's summary captures it: "coder effects are always about evolving a computation scheme." The word *evolving* is key. You do not start with a textbook algorithm and optimize it until it fits. You evolve the algorithm and the implementation together, each constraining and enabling the other, until you find a form that works within the hardware's budget.

Practical: A Simplified 56x56 Spinning Sphere

Let us sketch how you would build a simplified version of this effect. We will target a 56x56 pixel sphere — 7 bytes wide at the equator, 56 scanlines tall. The goal is not to reproduce Illusion's full rendering engine, but to understand the core technique well enough to implement it.

Step 1: Precompute the Sphere Geometry

For each scanline *y* (from -28 to +27, centered on the sphere), compute the visible arc:

```
radius_at_y = sqrt(R^2 - y^2)    ; where R = 28 (sphere radius in pixels)
```

This gives the half-width of the sphere at that scanline. For each pixel position *x* within that arc, compute the corresponding longitude and latitude on the sphere surface:

```
latitude  = arcsin(y / R)
longitude = arcsin(x / radius_at_y) + rotation_angle
```

These give the (u, v) coordinates into the source texture for each screen pixel.

Step 2: Build Skip Tables

Rather than storing full (u, v) pairs for every pixel (prohibitively expensive in memory), compute the *difference in source position* between adjacent screen pixels. For each scanline, you need a list of skip values: how many source pixels to advance between consecutive screen samples.

Near the equator, consecutive screen pixels map to nearly adjacent source pixels — skips of 1. Near the poles, the projection compresses, and you skip over more source pixels — skips of 2, 3, or more.

Store these as a table. For our 56x56 sphere, you need at most 56 entries per line (the widest line), times 56 lines, times one byte per entry. That is at most 3,136 bytes for a single rotation angle — but in practice, you can exploit vertical symmetry (top half mirrors bottom half) and store only half the table.

For animation, you need skip tables for multiple rotation angles. With 32 rotation steps, you could fit the tables into 32 x 1,568 = about 49KB. That overflows available memory, so in practice you would use fewer rotation steps, coarser angular resolution, or regenerate tables on the fly from a compact representation.

Step 3: Generate the Rendering Code

For each frame, read the skip table for the current rotation angle and generate Z80 code:

“‘z80 id:ch06_step_3_generate_the_rendering ; Code generator pseudocode (in Z80 assembly, this would be ; a loop that writes opcodes into a buffer)

generate_sphere_code: ld iy,skip_table ; pointer to skip distances ld ix,code_buffer ; pointer to output code buffer

.line_loop: ; For each scan line… ld b,bytes_this_line ; number of output bytes (e.g. 7 at equator)

.byte_loop: ; For each output byte, emit code for 8 pixels: ld c,8 ; 8 pixels per byte

.pixel_loop: ; Emit: ADD A,A ld (ix+0),$87 ; opcode for ADD A,A inc ix

```
; Emit: ADD A,(HL)
ld   (ix+0),$86           ; opcode for ADD A,(HL)
inc  ix

; Emit INC L instructions based on skip distance
ld   a,(iy+0)             ; read skip distance
inc  iy
```

.emit_inc_l: or a jr z,.pixel_done ld (ix+0),$2C ; opcode for INC L inc ix dec a jr nz,.emit_inc_l

.pixel_done: dec c jr nz,.pixel_loop

```
; Emit: LD (DE),A  (write byte to screen)
ld   (ix+0),$12           ; opcode for LD (DE),A
inc  ix
; Emit: INC E
ld   (ix+0),$1C           ; opcode for INC E
inc  ix
```

```
dec  b
jr   nz,.byte_loop

; Emit line transition code here (advance DE to next screen line)
; ...

jr   .line_loop


This is simplified --- the actual Illusion code is more tightly integrated, and Dark likely us

### Step 4: Execute and Display

Once the code buffer is filled, call it as a subroutine:

```z80 id:ch06_step_4_execute_and_display
    ld   hl,source_image      ; source texture (page-aligned, 1 byte/pixel)
    ld   de,screen_address    ; start of sphere area in video memory
    call code_buffer          ; execute the generated rendering code
```

The generated code runs through the entire sphere, reading source pixels, packing them into screen bytes, and writing them to video memory. When it returns, the sphere is drawn.

For animation, increment the rotation angle, load the corresponding skip table (or regenerate it), regenerate the code, and render again.

Step 5: Source Image Layout

The source texture must be organized for fast sequential access. Since the rendering code uses `INC L` to advance through it, the texture must be page-aligned (start at an address where the low byte is `$00`), and each row must fit within 256 bytes. A 256-pixel-wide texture stored as one byte per pixel fits this constraint perfectly: each row occupies one page.

For the monochrome case, each pixel is `$00` or `$01`. This means `ADD A,(HL)` either adds 0 (pixel off) or 1 (pixel on) to the accumulator's lowest bit, right after `ADD A,A` has shifted everything up. The result is a bit-packed screen byte where each bit corresponds to a sampled source pixel.

The Larger Pattern

The sphere in Illusion is a specific instance of a general demoscene pattern that appears throughout this book. The pattern has three parts:

Precomputation. Expensive mathematical work — projection, trigonometry, coordinate transforms — is done once (or once per frame) and stored as compact tables. The tables encode *what* to render without encoding *how*.

Code generation. The rendering code itself is generated from the tables. This eliminates branches, loop counters, and conditional logic from the inner loop.
```

Every instruction in the generated code does useful work. There is no overhead for "figuring out what to do next" — that decision was made during generation.

Sequential memory access. The inner loop reads data sequentially, advancing a pointer with single-byte increments. This is the fastest possible access pattern on the Z80, where register-indirect loads (`LD A,(HL)`) are cheap and indexed addressing (`LD A,(IX+d)`) is expensive.

The rotozoomer in the next chapter uses the same pattern. So does the dotfield scroller in Chapter 10. So do the attribute tunnels in Chapter 9. The specifics differ — different tables, different generated code, different data formats — but the architecture is the same. Introspec recognized this when he wrote that coder effects are about "evolving a computation scheme." The sphere, the rotozoomer, the tunnel: they are all evolved from the same fundamental approach. The evolution is in the details — which computation, which table layout, which inner loop — but the skeleton is shared.

Dark understood this in 1996. He encoded it in his Spectrum Expert articles in 1997. Introspec confirmed it by disassembly in 2017. The pattern is as valid now as it was then, on any platform where T-states are scarce and every instruction must earn its keep.

Summary

- The sphere effect in Illusion maps a monochrome source image onto a rotating sphere using dynamically generated Z80 code.
- Lookup tables encode the sphere's geometry as pixel skip distances. The rendering code is generated from these tables at runtime.
- The inner loop uses `ADD A,A` and `ADD A,(HL)` to accumulate pixels into screen bytes, with variable numbers of `INC L` instructions to advance through the source data.
- Performance: 101 + 32x T-states per output byte, where x depends on position.
- The approach exemplifies a general demoscene pattern: precompute geometry, generate code, access memory sequentially.
- Dark applied these algorithms in Illusion (1996), then documented them in Spectrum Expert (1997–98). Introspec reverse-engineered the result twenty years later, confirming the techniques.

Sources: Introspec, "Technical Analysis of Illusion by X-Trade" (Hype, 2017); Dark, "Programming Algorithms" (Spectrum Expert #01, 1997). The Hype comment thread includes contributions from kotsoft, Raider, and others.

Chapter 7: Rotozoomer and Chunky Pixels

> *"The trick is that you don't rotate the screen. You rotate your walk through the texture."* – paraphrasing the core insight behind every rotozoomer ever written

---

There is a moment in Illusion where the screen fills with a pattern – a texture, monochrome, repeating – and then it begins to turn. The rotation is smooth and continuous, the zoom breathes in and out, and the whole thing runs at a pace that makes you forget you are watching a Z80 push pixels at 3.5 MHz. It is not the most technically demanding effect in the demo. The sphere (Chapter 6) is harder mathematically. The dotfield scroller (Chapter 10) is tighter in its cycle budget. But the rotozoomer is the one that looks effortless, and on the Spectrum, making something look effortless is the hardest trick of all.

This chapter traces two threads. The first is Introspec's 2017 analysis of the rotozoomer from Illusion by X-Trade. The second is sq's 2022 article on Hype about chunky pixel optimisation, which pushes the approach to 4x4 pixels and catalogues a family of rendering strategies with precise cycle counts. Together, they map the design space: how chunky pixels work, how rotozoomers use them, and the performance trade-offs that determine whether your effect runs at 4 frames per screen or 12.

What a Rotozoomer Actually Does

A rotozoomer displays a 2D texture rotated by some angle and scaled by some factor. The naive approach: for every screen pixel, compute its corresponding texture coordinate via a trigonometric rotation:

```
tx = sx * cos(theta) * scale  +  sy * sin(theta) * scale  +  offset_x
ty = -sx * sin(theta) * scale  +  sy * cos(theta) * scale  +  offset_y
```

At 256x192, that is 49,152 pixels each needing two multiplications. Even with a 54-T-state square-table multiply (Chapter 4), you exceed five million T-states – roughly 70 frames' worth of CPU time. The effect is mathematically trivial and computationally impossible.

The key insight is that the transformation is *linear*. Moving one pixel right on screen always adds the same (dx, dy) to the texture coordinates. Moving one

pixel down always adds the same (dx', dy'). The per-pixel cost collapses from two multiplications to two additions:

```
Step right:   dx = cos(theta) * scale,   dy = -sin(theta) * scale
Step down:    dx' = sin(theta) * scale,  dy' = cos(theta) * scale
```

Start each row at the correct texture coordinate and step by (dx, dy) for every pixel. The inner loop becomes: read the texel, advance by (dx, dy), repeat. Two additions per pixel, no multiplications. The per-frame setup is four multiplications to compute the step vectors from the current angle and scale. Everything else follows from linearity.

This is the fundamental optimisation behind every rotozoomer on every platform. On the Amiga, on the PC, on the Spectrum.

Fixed-Point Stepping on the Z80

On a 16-bit or 32-bit platform, dx and dy would be fixed-point values: the integer part selects the texel, and the fractional part accumulates sub-pixel precision. On the Z80, we lack the registers and the bandwidth for true fixed-point inner loops. The classic Spectrum solution is to collapse the step to integer increments – always exactly +1, -1, or 0 per axis – and control the *ratio* of steps between axes to approximate the angle.

Consider a rotation of 30 degrees. The exact step vector would be (cos 30, -sin 30) = (0.866, -0.5). On a machine with fixed-point arithmetic, you would add 0.866 to the column coordinate and subtract 0.5 from the row coordinate per pixel. On the Z80, the inner loop instead alternates between two integer steps: some pixels step (+1 column, 0 rows) and others step (+1 column, -1 row). If you distribute these in a roughly 2:1 ratio – two column-only steps for every diagonal step – the average direction approximates the 0.866:0.5 ratio of a 30-degree walk. This is Bresenham's line algorithm applied to texture traversal.

The zoom factor determines how many texels you skip per screen pixel. At scale 1.0, every texel maps to one screen pixel. At scale 2.0, you skip every other texel, effectively zooming in. On the Spectrum, this is controlled by doubling the walk instructions: instead of one `INC L` per pixel, you execute two, stepping by 2 texels and producing a 2x zoom. Intermediate zoom levels again use Bresenham-like distribution: some pixels step by 1, others by 2, with the ratio controlled by an error accumulator.

The per-frame cost of computing these parameters is negligible: four lookups into a sine table, a few multiplications (or table lookups, see Chapter 4), and a Bresenham setup pass. All the heavy work is in the inner loop, which has been reduced to nothing but register increments and memory reads.

Chunky Pixels: Trading Resolution for Speed

Even at two additions per pixel, writing 6,144 bytes to the Spectrum's interleaved video memory per frame is impractical – not if you also want to update the angle and leave time for music. Chunky pixels solve this by reducing the effective resolution. Instead of one texel per screen pixel, you map one texel to a 2x2, 4x4, or 8x8 block.

Illusion uses 2x2 chunky pixels: effective resolution 128x96, a 4x reduction in work. The effect looks blocky up close, but at the speed the texture sweeps across the screen, motion hides the coarseness. The eye forgives low resolution when everything is moving.

Why 2x2 Is the Sweet Spot

The choice of chunk size involves a three-way tradeoff: visual quality, rendering speed, and memory. At 2x2, you get 128x96 effective pixels – enough to read text and recognise patterns in the texture. At 4x4, the 64x48 grid is noticeably coarser; fine details in the texture become unreadable, but the effect still "reads" as a coherent rotating surface. At 8x8, you are down to 32x24 blocks, which is the attribute grid resolution – any texture detail is lost, and the effect looks like coloured rectangles. The last case can be useful for colour-only effects (attribute tunnels, Chapter 9), but for a pixel-rendered rotozoomer, 2x2 or 4x4 is the practical range.

The memory cost matters too. Each chunky pixel stores one byte, so a 2x2 rotozoomer at 128x96 needs 12,288 texels per frame. With a 256-byte texture row (the natural width for 8-bit wrapping), the texture itself occupies 256 bytes per row times however many rows you need. A 4x4 version only processes 3,072 texels, which means the inner loop runs one-quarter as many iterations – but the visual cost is significant.

In practice, Spectrum demos land on 2x2 for featured rotozoomer effects and reserve 4x4 for situations where the rotozoomer shares the screen with other effects (bumpmapping overlays, split-screen compositions).

The $03 Encoding Trick

The encoding is designed for the inner loop. Each chunky pixel is stored as `$03` (on) or `$00` (off). This value is not arbitrary – it encodes exactly the two low bits set: `%00000011`. Watch what happens as four pixels accumulate in the A register:

```
After pixel 1:  A = %00000011                  ($03)
After 2x shift: A = %00001100                  ($0C)
After pixel 2:  A = %00001100 + %00000011      ($0F)
After 2x shift: A = %00111100                  ($3C)
After pixel 3:  A = %00111100 + %00000011      ($3F)
After 2x shift: A = %11111100                  ($FC)
After pixel 4:  A = %11111100 + %00000011      ($FF)
```

If all four pixels are "on", the result is `$FF` – all bits set. If all four are "off" (`$00`), the shifts and additions produce `$00`. Mixed patterns produce the correct 2-bit-per-pixel stripe: for example, on-off-on-off gives `%11001100` = `$CC`. Each pair of bits in the output byte corresponds to one chunky pixel. Since each chunky pixel is 2 screen pixels wide (2x2), the 8-bit output byte covers exactly 8 screen pixels: four chunky columns times two pixels each.

The critical property: because we only ever add `$03` or `$00`, there is no carry between pixel fields. The two-bit groups never overflow into each other. This is what makes the encoding branchless – no masking needed, no OR operations, just `ADD A,A` and `ADD A,(HL)`.

The Inner Loop from Illusion

Introspec's disassembly reveals the core rendering sequence. HL walks through the texture; H tracks one axis and L the other:

```
z80 id:ch07_the_inner_loop_from_illusion ; Inner loop: combine 4 chunky pixels
into one output byte     ld   a,(hl)        ;  7T  -- read first chunky pixel
($03 or $00)     inc  l             ;  4T  -- step right in texture     dec  h
;  4T  -- step up in texture     add  a,a           ;  4T  -- shift left     add
a,a           ;  4T  -- shift left (now shifted by 2)     add  a,(hl)        ;
7T  -- add second chunky pixel
```

The sequence repeats for the third and fourth pixels. The `inc l` and `dec h` together trace a diagonal path through the texture – and diagonal means rotated. The specific combination of increment and decrement instructions determines the rotation angle.

Step	Instructions	T-states
Read pixel 1	`ld a,(hl)`	7
Walk	`inc l : dec h`	8
Shift + Read pixel 2	`add a,a : add a,a : add a,(hl)`	15
Walk	`inc l : dec h`	8
Shift + Read pixel 3	`add a,a : add a,a : add a,(hl)`	15
Walk	`inc l : dec h`	8
Shift + Read pixel 4	`add a,a : add a,a : add a,(hl)`	15
Walk	`inc l : dec h`	8
Output + advance	`ld (de),a : inc e`	~11
Total per byte		**~95**

Introspec measured approximately 95 T-states per 4 chunks.

The critical observation: the walk direction is hardcoded into the instruction stream. A different rotation angle requires different instructions. Eight primary directions are possible using combinations of `inc l`, `dec l`, `inc h`, `dec h`, and `nop`. This means the rendering code changes every frame.

Self-Modifying Code at the Byte Level

"Per-frame code generation" sounds exotic, but the mechanism is mundane. Each walk instruction is a single byte in memory. `INC L` is opcode `$2C`. `DEC L` is `$2D`. `INC H` is `$24`. `DEC H` is `$25`. `NOP` is `$00`. To change the walk direction from "right and up" (`INC L` + `DEC H`) to "pure right" (`INC L` + `NOP`), you write `$00` to the byte where `$25` currently sits. That is the entire code generation step: `LD A,$00 : LD (walk_target),A`. A few stores into the instruction stream, and the inner loop now walks in a different direction.

The targets are known at assembly time. Each SMC site is labelled (e.g., `.smc_walk_h_0:`) and the patching code uses those labels as literal addresses. There is no dynamic memory allocation, no instruction parsing, no runtime disassembly.

You are writing known opcodes to known addresses. The Z80 has no instruction cache to invalidate, no pipeline to flush. The write takes effect immediately on the next fetch from that address.

In a fully unrolled inner loop (which Illusion uses for its 16-byte rows), there would be 64 walk-instruction sites to patch: 4 walk pairs per output byte times 16 bytes per row. Patching 64 bytes costs about 64 x 13 = 832 T-states (each `LD (nn),A` is 13 T-states), which is negligible compared to the 100,000+ T-states the rendering pass takes. The code generator is cheap. The generated code is what matters.

Per-Frame Code Generation

The rendering code is generated fresh each frame, with walk-direction instructions patched for the current angle:

Angle range	H step	L step	Direction
~0 degrees	`nop`	`inc l`	Pure right
~45 degrees	`dec h`	`inc l`	Right and up
~90 degrees	`dec h`	`nop`	Pure up
~135 degrees	`dec h`	`dec l`	Left and up
~180 degrees	`nop`	`dec l`	Pure left
~225 degrees	`inc h`	`dec l`	Left and down
~270 degrees	`inc h`	`nop`	Pure down
~315 degrees	`inc h`	`inc l`	Right and down

For intermediate angles, the generator distributes steps unevenly using Bresenham-like error accumulation. A 30-degree rotation alternates between `inc l : nop` and `inc l : dec h` at roughly a 2:1 ratio, approximating the 1.73:1 tangent of 30 degrees. The resulting code is an unrolled loop where each iteration has its own specific walk pair, tuned to the current angle.

The rendering cost for 128x96 at 2x2 chunky. The 128x96 area is 96 pixel rows, but each 2x2 texel covers two pixel rows, giving 48 texel rows. Each texel row produces 16 output bytes (128 pixels / 8 bits per byte, with 4 chunky pixels packed per byte):

```
16 output bytes/row x 95 T-states = 1,520 T-states/row
1,520 x 48 texel rows = 72,960 T-states total
```

Roughly 1 frame on a Pentagon (71,680 T-states per frame). But this is the bare inner loop only. A complete accounting adds:

```
Code generation:        ~  1,000 T  (patching walk instructions)
Row setup (per row):    ~    800 T  (48 rows x ~17 T each)
Buffer-to-screen copy:  ~ 20,000 T  (stack trick, 1,536 bytes)
Sine table lookups:     ~    200 T
Frame overhead:         ~    500 T  (HALT, border, angle update)
                        ----------
Inner loop:               72,960 T
Total per frame:        ~ 95,460 T  (= 1.33 Pentagon frames)
```

On a standard 48K/128K Spectrum at 69,888 T-states per frame, the rendering takes roughly 1.4 frames. Introspec's estimate of 4-6 frames per screen accounts for the more complex code path in Illusion (which handles the full 256x192 screen, not just a 128x96 strip) and the cost of the music engine running in the interrupt. On a Pentagon with its slightly longer frame (71,680 T-states) and no contention, the inner loop runs about 3% faster.

Memory contention on the 48K/128K Spectrum adds another hidden cost. During the top 192 scanlines, the ULA steals cycles from the CPU when accessing the lower 16KB of RAM ($4000-$7FFF). The inner loop reads from the texture (which should be above $8000, out of contended memory) and writes to a buffer (also above $8000), so it avoids contention entirely. The buffer-to-screen transfer, however, writes directly to video RAM and will be slowed by contention if it overlaps with the display period. This is why demos synchronise the screen transfer to the border period or to the bottom of the display.

Buffer to Screen Transfer

The rotozoomer renders into an off-screen buffer, then transfers to video memory. The interleaved screen layout makes direct rendering painful, and buffering avoids tearing.

The transfer uses the stack:

```
z80 id:ch07_buffer_to_screen_transfer     pop  hl                    ; 10T --
read 2 bytes from buffer     ld   (screen_addr),hl     ; 16T -- write 2 bytes to
screen
```

Screen addresses are embedded as literal operands, pre-calculated for the Spectrum's interleaving – another instance of code generation. At 26 T-states per two bytes, a full 1,536-byte transfer costs under 20,000 T-states. The rendering pass is the bottleneck, not the transfer.

Deep Dive: 4x4 Chunky Pixels (sq, Hype 2022)

sq's article pushes chunky pixels to 4x4 – effective resolution 64x48. The visual result is coarser, but the performance gain opens up effects like bumpmapping and interlaced rendering. The article is a study in optimisation methodology: start straightforward, iteratively improve, measure at each step.

Approach 1: Basic LD/INC (101 T-states per pair). Load chunky value, write to buffer, advance pointers. The bottleneck is pointer management: `INC HL` at 6 T-states adds up over thousands of iterations.

Approach 2: LDI variant (104 T-states – slower!). `LDI` copies a byte and auto-increments both pointers in one instruction. But it also decrements BC, consuming a register pair. The save/restore overhead makes it *slower* than the naive approach. A cautionary tale: on the Z80, the "clever" instruction is not always the fast one.

Approach 3: LDD dual-byte (80 T-states per pair). By arranging source and destination in reverse order, `LDD`'s auto-decrement works in your favour. A combined two-byte sequence exploits this for a 21% improvement over baseline.

Approach 4: Self-modifying code (76-78 T-states per pair). Pre-generate 256 rendering procedures, one per possible byte value, each with the pixel value baked in as an immediate operand:

```
z80 id:ch07_deep_dive_4x4_chunky_pixels ; One of 256 pre-generated procedures
proc_A5:     ld   (hl),$A5        ; 10T  -- value baked into instruction     inc
l               ;  4T     ld   (hl),$A5        ; 10T  -- 4x4 block spans 2 bytes
horizontally     ; ... handle vertical repetition ...     ret                  ;
10T
```

The 256 procedures occupy approximately 3KB. Per-pixel rendering drops to 76-78 T-states – 23% faster than baseline, 27% faster than LDI.

Performance Comparison

Approach	Cycles/pair	Relative	Memory
Basic LD/INC	101	1.00x	Minimal
LDI variant	104	0.97x	Minimal
LDD dual-byte	80	1.26x	Minimal
Self-modifying (256 procs)	76-78	1.30x	~3KB

The self-modifying approach wins, but the margin over LDD is narrow. In a 128K demo, 3KB is easily available. In a 48K production, the LDD approach might be the better engineering decision.

Historical Roots: Born Dead #05 and the Scene Lineage

sq notes these techniques build on work published in Born Dead #05, a Russian demoscene newspaper from approximately 2001. Born Dead was one of several Russian-language disk magazines that served as technical journals for the ZX Spectrum demoscene. Unlike Western PC demoscene publications that could assume 486-class hardware, the Spectrum magazines operated under the constraints of a community that was still actively developing new techniques for a machine from 1982. The foundational article described basic chunky rendering – the idea that you could treat the Spectrum's bit-mapped display as a lower-resolution chunky-pixel buffer and gain speed at the expense of resolution.

sq's contribution, twenty-one years later, was the systematic optimisation and the pre-generated procedure variant. But between Born Dead #05 and sq's 2022 article, the chunky rotozoomer appeared in numerous Spectrum demos. X-Trade's Illusion (ENLiGHT'96) was among the earliest full implementations. Other notable examples include Exploder^XTM's GOA4K and Refresh, 4D's productions, and later work from the Russian and Polish scenes. The technique spread partly through disassembly – Introspec's 2017 analysis of Illusion is itself an example of the scene's tradition of learning by reverse engineering – and partly through the informal knowledge network of disk magazines, BBS postings, and direct communication between coders.

This is how scene knowledge evolves: a technique surfaces in an obscure disk magazine, circulates within the community, and twenty-one years later someone revisits it with fresh measurements and new tricks. The chain from Born Dead to sq to this chapter is unbroken.

Practical: Building a Simple Rotozoomer

Here is the structure for a working rotozoomer with 2x2 chunky pixels and a checkerboard texture.

Texture. A 256-byte page-aligned table where each byte is `$03` or `$00`, generating 8-pixel-wide stripes. The H register provides the second dimension; XORing H into the lookup creates a full checkerboard:

```
lua id:ch07_practical_building_a_simple     ALIGN 256 texture:     LUA ALLPASS
for i = 0, 255 do         if math.floor(i / 8) % 2 == 0 then             sj.add_byte(0x03)
else             sj.add_byte(0x00)         end     end     ENDLUA
```

Sine table and per-frame setup. A 256-entry page-aligned sine table drives the rotation. Each frame reads `sin(frame_counter)` and `cos(frame_counter)` (cosine via a 64-index offset) to compute the step vectors, then patches the inner loop's walk instructions with the correct opcodes.

The rendering loop. The outer loop sets the starting texture coordinate for each row (stepping perpendicular to the walk direction). The inner loop walks through the texture:

```
z80 id:ch07_practical_building_a_simple_2 .byte_loop:     ld   a,(hl)             ;
read texel 1     inc  l                   ; walk (patched per-frame)     add
a,a : add  a,a     ; shift     add  a,(hl)              ; read texel 2     inc
l                   ; walk     add  a,a : add  a,a     ; shift     add  a,(hl)
; read texel 3     inc  l                   ; walk     add  a,a : add  a,a     ;
shift     add  a,(hl)             ; read texel 4     inc  l                   ;
walk     ld   (de),a             ; write output byte     inc  de     djnz
.byte_loop
```

The `inc l` instructions are the targets of the code generator. Before each frame, they are patched to the appropriate combination of `inc l`/`dec l`/`inc h`/`dec h`/`nop` based on the current angle. For non-cardinal angles, a Bresenham error accumulator distributes the minor-axis steps across the row, so each walk instruction in the unrolled loop may be different from its neighbours.

Main loop. `HALT` for vsync, compute step vectors, generate walk code, render to buffer, stack-copy buffer to screen, increment frame counter, repeat.

Texture Design and Boundary Handling

The texture is the most constrained data structure in the rotozoomer. Every design decision in the inner loop – the page alignment, the wrapping behaviour, the power-of-two sizing – traces back to how the texture is laid out in memory.

Why Page-Aligned, Why 256 Columns

The texture is page-aligned so that H selects the row and L selects the column. This is not merely convenient; it makes the inner loop possible. `INC L` and `DEC L` wrap at the 256-byte page boundary automatically – when L overflows from `$FF` to `$00`, H is unchanged. The texture wraps horizontally for free, with zero branch overhead. If the texture were not page-aligned, L increments would carry into H, corrupting the row address. You would need explicit masking (`AND $3F` after every step), which would add 4-8 T-states per pixel and destroy the tight inner loop.

The vertical axis (H) also wraps, but over the full range of rows allocated to the texture. If you allocate 64 rows (pages), H ranges from the texture base page to base+63. `INC H` and `DEC H` will happily walk past the end of the texture into whatever memory follows. Illusion handles this by masking H to the texture height at the start of each row (not per pixel – per-pixel masking would be too expensive). This works because within a single 16-byte row, the H coordinate changes by at most 16 steps, and if the texture is tall enough relative to the row width, an overflow within a row cannot reach memory that produces visual garbage. A 64-row texture with 16 H-steps per row has a comfortable margin.

Choosing Texture Size

The texture must be a power-of-two in width (always 256, since L is 8 bits) and ideally a power-of-two in height for easy masking. Common choices:

- **256x256** (64KB): fills all of upper RAM on a 128K Spectrum. Maximum resolution, but leaves no room for code or buffers.
- **256x64** (16KB): the practical choice. Fits in one 16KB bank on 128K hardware. The 6-bit height mask (`AND $3F`) is fast and tiles seamlessly.
- **256x32** (8KB): fits on a 48K Spectrum with room for everything else. The texture repeats more visibly, but for a checkerboard or stripe pattern, repetition *is* the design.
- **256x16** (4KB): minimal. Works for very simple patterns like single-axis stripes.

For non-repeating textures (images, logos), the height should be at least as large as the effective screen height divided by the scale factor. A 2x2 rotozoomer with 96 effective rows needs at least 96 texture rows to avoid visible tiling when the zoom is at 1:1. At higher zoom levels, fewer rows are needed because the camera is “closer” to the texture surface.

What About Screen Boundaries?

The Spectrum’s 256x192 screen is 32 bytes wide by 192 lines. If your rotozoomer fills a 128x96 strip in the centre, you never approach the edge of video memory. But a full-screen rotozoomer at 256x192 (or even 128x192 with 2x2 chunky) must handle the case where the output address reaches the attribute area at `$5800`. The simplest approach: render into a buffer and only copy the portion that fits. A more aggressive approach: clip the row count to the visible area during code generation, which avoids wasted computation but adds complexity to the row loop.

In practice, most Spectrum rotozoomers render a strip smaller than the full screen. The visual framing – a border, a title bar, a music credit – hides the cropping and buys back T-states for other effects.

The Design Space

The chunky pixel size is the most consequential design decision in a rotozoomer:

Parameter	2x2 (Illusion)	4x4 (sq)	8x8 (attributes)
Resolution	128x96	64x48	32x24
Texels/frame	12,288	3,072	768
Inner loop cost	~73,000 T	~29,000 T	~7,300 T
Frames/screen	~1.3	~0.5	~0.1
Visual quality	Good motion	Chunky but fast	Very blocky
Use case	Featured effects	Bumpmapping, overlays	Attribute-only FX

The 4x4 version fits within a single frame with room for a music engine and other effects. The 2x2 version takes roughly 1.3-1.5 frames (including overhead) but looks substantially better. The 8x8 case is the attribute tunnel from Chapter 9.

Once you have a fast chunky renderer, the rotozoomer is just one application. The same engine drives **bumpmapping** (read height differences instead of raw texels, derive shading), **interlaced effects** (render odd/even rows on alternating frames, doubling effective frame rate at the cost of flicker), and **texture distortion** (vary the walk direction per row for wavy or ripple effects). A 4x4 rotozoomer can share a frame with a scrolltext, a music engine, and a screen transfer. sq's work was motivated by exactly this versatility.

Three Approaches to Texture Rotation

Everything above treats the rotozoomer as one technique with a tuneable chunk size. But "rotozoomer" on the Spectrum is really a family of three distinct approaches, each with different inner loops, different visual character, and different performance profiles. They share the same mathematical foundation – the linear step vectors, the Bresenham-style angle distribution – but diverge completely at the rendering level.

Variant 1: Monochrome Bitmap (Full Pixel Resolution)

The purest form: every screen pixel maps to one texel. The texture is monochrome – one bit per pixel – so reading a texel means testing a single bit, and writing to the screen means setting or clearing a single bit. No chunky encoding, no block grouping. The result is a rotated texture at the full 256x192 resolution of the Spectrum display.

The inner loop skeleton looks something like this:

```
z80 id:ch07_variant_1_monochrome_bitmap ; For each screen pixel: ; DE = texture
pointer, HL = screen pointer     ld   a,(de)           ;  7T  -- read texture
byte     and  n                ;  7T  -- test texture bit at current coords
jr   z,.pixel_off     ; 12/7T     set  m,(hl)           ; 15T  -- set screen
bit     jr   .pixel_done      ; 12T .pixel_off:     res  m,(hl)           ; 15T
-- clear screen bit .pixel_done:     ; advance texture coords (inc e / dec d /
etc.)     ; advance screen bit position     ; ... next pixel
```

Note that SET and RES only work with (HL), (IX+d), or (IY+d) – not (DE) or (BC). This forces HL to serve as the screen pointer, while DE handles the texture coordinates.

The per-pixel cost is brutal: 35-45 T-states minimum, with branching on every pixel. Across 49,152 pixels, that is 1.5 to 2 million T-states for the rendering pass alone – roughly 21-28 frames on a standard Spectrum. A full-screen monochrome rotozoomer at 50fps is not happening.

But nobody said you need to fill the whole screen. The technique shines when applied to a smaller region – a 128x64 strip, a circular viewport, a masked area – or when you accept a lower frame rate in exchange for the visual impact of full-resolution rotation. It also works beautifully for distortion effects where the "rotation" is not uniform: varying the step vectors per scanline produces wave distortions, barrel effects, and the "sonic ripple" look seen in parts of Illusion by Dark/X-Trade. The coordinate mapping is no longer a simple rotation but a per-line warp through the texture. The maths is the same – fixed-point stepping along a direction – but the direction itself changes every row.

The visual payoff is striking. Where a 2x2 chunky rotozoomer looks like a rotating mosaic, the monochrome bitmap version looks like a rotating *image*. On a machine where every effect fights the same 69,888 T-state budget, dedicating multiple frames to full-resolution rendering is a deliberate aesthetic choice.

Variant 2: Chunky Rotozoomer (2x2 or 4x4 Blocks)

This is the technique covered in the bulk of this chapter. Each screen block (2x2 or 4x4 pixels) maps to one texel. The `$03`/`$00` encoding, the `add a,a : add a,(hl)` accumulation, the walk-instruction patching – all of it targets this approach.

At 2x2 (128x96 effective resolution), the inner loop runs at approximately 95 T-states per output byte, producing the smooth, recognisable rotozoomer seen in Illusion. At 4x4 (64x48), sq's pre-generated procedure variant eliminates the loop overhead entirely, bringing the cost down to 76-78 T-states per output pair and leaving room for multi-effect compositions within a single frame.

The chunky rotozoomer occupies the middle ground: fast enough for real-time, detailed enough to carry a featured effect. It is the workhorse of the Spectrum rotozoomer repertoire.

Variant 3: Attribute Rotozoomer (8x8 Block "Pixels")

The Spectrum's attribute area at `$5800`-`$5AFF` stores colour information for each 8x8 pixel character cell: 32 columns by 24 rows, 768 bytes total. Each byte encodes INK, PAPER, BRIGHT, and FLASH for a single 8x8 block. The attribute rotozoomer

ignores the bitmap entirely and treats these 768 attribute cells as the display surface. Each cell becomes one "pixel" in a 32x24 image.

The inner loop is structurally identical to the chunky version – step through texture coordinates, read a value, write it to the output – but the output is the attribute area, and the "texel" value is a colour attribute byte rather than a bit pattern. The effective resolution is just 32x24, which means the entire rendering pass is 768 iterations of the stepping loop.

The maths:

```
32 columns x 24 rows = 768 attribute cells
~10 T-states per cell (read texel + write attribute + step)
768 x 10 = ~7,680 T-states total
```

That is roughly 11% of a single frame. You could run the attribute rotozoomer nine times over and still have room for a music engine. The cost is so low that the effect is essentially free.

But the visual payoff is different from the bitmap variants. You are not rotating pixels – you are rotating coloured blocks. At 32x24, fine detail is invisible. What you get instead is a sweeping field of colour, a vivid mosaic that turns and breathes. The attribute rotozoomer in Illusion uses exactly this: a boldly coloured texture (not a monochrome bitmap) mapped through the attribute grid, producing the characteristic "stained glass" look of rotating colour fields that Illusion is known for. The PAPER and INK fields in each attribute byte give you two colours per cell, so a carefully designed texture can pack more visual information than the raw resolution suggests.

The attribute rotozoomer is perfect for backgrounds, transitions, or as a base layer with pixel-level effects composited on top. Because it only writes to the attribute area, the bitmap can be used simultaneously for a different effect – a scroller, a logo, a particle field – running at its own pace. This layered approach is a hallmark of multi-effect demo screens on the Spectrum.

Comparison

Vari-ant	Effective resolution	Bytes written/frame	~T-states (render)	Colour	Typical use
Monochrome bitmap	256x192 (or subregion)	6,144 (full screen)	1,500,000-2,000,000	1-bit	Hero effect, distortion, warp
Chunky 2x2	128x96	1,536	~73,000	1-bit	Featured roto-zoomer
Chunky 4x4	64x48	384	~29,000	1-bit	Multi-effect, overlay

Vari-ant	Effective resolution	Bytes written/frame	~T-states (render)	Colour	Typical use
At-tribute	32x24	768	~7,700	INK+PA-PER (2 colours/cell)	Back-ground, colour wash, transition

The progression from top to bottom is a smooth trade: resolution for speed, detail for headroom. The monochrome bitmap gives you everything the Spectrum's display can show, at a cost that demands dedication. The attribute version gives you almost nothing in resolution, but it runs so fast that the rotozoomer becomes just another instrument in a multi-effect composition rather than the main event.

All four rows in this table share the same core algorithm. The step vectors are computed the same way. The Bresenham distribution works the same way. The difference is only where you write and how many iterations you run. Once you have built one rotozoomer, you have built all of them.

The Rotozoomer in Context

The rotozoomer is not a rotation algorithm. It is a *memory traversal pattern*. You walk through a buffer in a straight line, and the walk direction determines what you see. Rotation is one choice of direction. Zoom is a choice of step size. The Z80 does not know trigonometry. It knows `INC L` and `DEC H`. Everything else is the programmer's interpretation.

In Illusion, the rotozoomer sits alongside the sphere and the dotfield scroller. All three share the same architecture: precomputed parameters, generated inner loops, sequential memory access. The sphere uses skip tables and variable `INC L` counts. The rotozoomer uses direction-patched walk instructions. The dotfield uses stack-based address tables. Three effects, one engine philosophy.

Dark built all of them. Introspec traced all of them. The pattern that connects them is the lesson of Part II: compute what you need before the inner loop starts, generate code that does nothing but read-shift-write, and keep the memory access sequential.

Summary

- A rotozoomer displays a rotated and zoomed texture by walking through it at an angle. Linearity reduces per-pixel cost from two multiplications to two additions.
- Chunky pixels (2x2, 4x4) reduce effective resolution and rendering cost proportionally. Illusion uses 2x2 at 128x96; sq's system uses 4x4 at 64x48.
- Illusion's inner loop: `ld a,(hl) : add a,a : add a,a : add a,(hl)` with walk instructions between reads. Cost: ~95 T-states per byte for 4 chunky pixels.

- Walk direction changes per frame, requiring code generation – the rendering loop is patched before each frame.
- sq's 4x4 optimisation journey: basic LD/INC (101 T-states) to LDI (104 T-states, slower) to LDD (80 T-states) to self-modifying code with 256 pre-generated procedures (76-78 T-states, ~3KB). Based on earlier work in Born Dead #05 (~2001).
- Buffer-to-screen transfer via `pop hl : ld (nn),hl` at ~26 T-states per two bytes.
- The rotozoomer shares its architecture with the sphere (Chapter 6) and dotfield (Chapter 10): precomputed parameters, generated inner loops, sequential memory access.

Sources: Introspec, "Technical Analysis of Illusion by X-Trade" (Hype, 2017); sq, "Chunky Effects on ZX Spectrum" (Hype, 2022); Born Dead #05 (~2001, original chunky pixel techniques).

Chapter 8: Multicolor — Breaking the Attribute Grid

"The multicolor will be conquered." — DenisGrachev, Hype, 2019

Every ZX Spectrum coder knows the rule. Two colours per 8x8 cell. Ink and paper. That is what the ULA gives you, and that is all it gives you. If your character has a red hat and a blue coat and the two colours fall inside the same attribute cell, one of them loses. The result — garish fringing, miscoloured sprites, characters that change colour as they walk past scenery — is attribute clash, and it is the defining visual constraint of the platform.

Attribute clash is so fundamental to the Spectrum's identity that many coders simply accept it. They design around it. They pick their palettes to minimise it. They restrict sprite sizes or avoid certain colour combinations. For thirty years, the 8x8 grid has been a fact of life.

But the ULA does not know this.

The ULA reads attribute bytes as it draws the screen, one scanline at a time. It does not read all 768 attribute bytes at once. It reads each row of 32 attributes exactly when it needs them, eight scanlines later it reads the same row again for the next pixel line within that character row, and so on. The attribute for any given cell is read eight times per frame — once for each pixel row in that cell.

The trick is obvious once you see it: if you change the attribute byte between reads, the ULA will apply a different colour to different pixel rows within the same cell. Instead of two colours for all eight rows, you get two colours per *group of rows*. The 8x8 grid does not break because the hardware was redesigned. It breaks because you rewrote the data faster than the hardware could consume it.

This is multicolor. It has been known since at least the early 2000s, when the Russian ZX magazine Black Crow published an algorithm and example code in its fifth issue. But for years, multicolor remained a curiosity — impressive in demos, impractical in games, because the CPU spent so many T-states changing attributes that nothing was left for game logic.

Then DenisGrachev figured out how to make games with it.

The ULA's Perspective

To understand multicolor, you need to see the screen from the ULA's point of view.

The ULA draws 192 visible scanlines per frame, top to bottom. Each scanline takes 224 T-states (on Pentagon). For each scanline, the ULA reads 32 pixel bytes and 32 attribute bytes from memory. The pixel bytes determine which dots are ink and which are paper. The attribute bytes determine which colours "ink" and "paper" actually are.

Within a character row (8 pixel lines), the ULA reads the same 32 attribute bytes for every scanline. It does not cache them — it reads them fresh each time. This means you have a window of opportunity between one scanline's attribute read and the next to change the attribute data.

The "traditional" multicolor approach exploits this directly. After a HALT (which synchronises the CPU with the frame interrupt), you count T-states to know exactly when the ULA will read each attribute row. Then, in the gap between reads, you overwrite the attribute bytes with new values. When the ULA reads them for the next scanline, it sees the new colours.

The limitation is brutal: counting T-states precisely, changing 32 bytes between scanlines, then waiting for the next opportunity. The CPU spends almost all its time on this bookkeeping. In a typical traditional multicolor engine, you can change attributes every 2 or 4 scanlines, giving 8x2 or 8x4 colour resolution. But the cycle budget consumed by the attribute-changing code itself leaves almost nothing for game logic, sprite rendering, or sound.

This is why multicolor stayed in demos. Demos can afford to spend 100% of the CPU on visual effects. Games cannot.

The LDPUSH Insight

In January 2019, DenisGrachev published an article on Hype titled "Multicolor Will Be Conquered" (Mul'tikolor budet pobezhdon). The title was a statement of intent. He had been developing Old Tower, a game for the ZX Spectrum with 8x2 multicolor — attributes changing every two pixel rows — and he wanted to explain how he had solved the cycle budget problem that made multicolor impractical in games.

The key insight is one of those ideas that seems inevitable in hindsight: the code that outputs pixel data *is* the display buffer.

Traditional multicolor separates code and data. You have a buffer of attribute bytes somewhere in memory, and rendering code that copies them to the screen at the right moment. DenisGrachev's technique fuses the two. The "buffer" is a sequence of Z80 instructions — specifically, `LD DE,nn` followed by `PUSH DE` — and executing those instructions writes the display data directly to screen memory via the stack pointer.

Here is how it works.

LD DE,nn / PUSH DE

The instruction `LD DE,nn` loads a 16-bit immediate value into register pair DE. It takes 10 T-states and is 3 bytes long: the opcode byte `$11`, followed by two data bytes (the value to load, low byte first). The instruction `PUSH DE` decrements SP by 1, writes the high byte of DE, decrements SP by 1 again, then writes the low byte. The result: SP ends up 2 lower, with the high byte at the higher address and the low byte at the lower address. It takes 11 T-states.

Together, `LD DE,nn : PUSH DE` costs 21 T-states, is 4 bytes long, and writes 2 bytes of data to the screen. The "data" is the immediate operand of the LD instruction. To change what gets drawn, you do not rewrite a display buffer — you patch the operand bytes inside the LD instructions themselves.

```
z80 id:ch08_ld_de_nn_push_de ; One LDPUSH pair: writes 2 bytes to screen memory
ld   de,$AA55       ; 10 T  load pixel data     push de             ; 11 T
write to (SP), SP = SP - 2                          ; ---                          ;
21 T total, 2 bytes output
```

A scanline of pixel data is 32 bytes wide. But you cannot fill all 32 bytes with PUSH alone, because PUSH writes downward (SP decrements) and you need the data to appear left-to-right on screen. The Spectrum's screen memory layout handles this: within a single scanline, consecutive bytes are at ascending addresses. PUSH writes to descending addresses. So the data comes out backwards — the last byte pushed appears at the lowest address, which is the leftmost byte on screen.

This means you build your LDPUSH sequence in reverse display order. The first `LD DE,nn : PUSH DE` in the code writes the rightmost two bytes of the scanline. The last writes the leftmost two bytes. When executed, the pushes fill the scanline right-to-left.

How much fits in a scanline?

DenisGrachev does not fill all 32 bytes. The GLUF engine (which powers Old Tower and other games) uses a 24-character-wide game area, padded with a border on each side. That is 24 bytes per scanline in the game area.

At 4 bytes of code per 2 bytes of output, you need 48 bytes of code to fill 24 bytes of screen. But there is an additional byte for the initial `LD SP,nn` setup and for the attribute change between scanline groups. DenisGrachev reports 51 bytes per scanline of generated code.

The beauty of this approach: there is no separate rendering pass. The instructions that populate the screen ARE the executable code. When you need to update a tile or sprite, you patch the immediate operands of the LD instructions. When the display code runs, it outputs the patched data. Code is data. Data is code.

The stack pointer as a cursor

During execution, SP is pointed at the right edge of the current scanline in screen memory. Each PUSH writes 2 bytes and decrements SP by 2, moving the "write cursor" leftward across the scanline. At the end of one scanline's output, SP points to the left edge. The code then adjusts SP to the right edge of the next scanline and repeats.

The fundamental constraint: interrupts must be disabled while SP is hijacked. If an interrupt fired, the CPU would push the return address into screen memory, corrupting the display. This means the multicolor rendering code runs with `DI` and re-enables interrupts with `EI` only after SP is restored to the real stack. The entire rendering pass — all 192 visible scanlines — happens in one uninterruptible block.

The GLUF Engine: Multicolor in a Real Game

Old Tower was a proof of concept. GLUF (DenisGrachev's game framework) was the production engine. Here are the numbers:

Parameter	Value
Multicolor resolution	8x2 (attributes change every 2 pixel rows)
Game area	24x16 characters (192x128 pixels)
Buffering	Double-buffered (two sets of display code)
Sprite size	16x16 pixels
Tile size	16x16 pixels
Sound	25 Hz (every other frame)
Cycles per frame	~70,000 (nearly the entire Pentagon budget)

The 8x2 resolution means the engine changes attributes four times per character row instead of once. Each character row is 8 scanlines tall; changing attributes every 2 scanlines gives four distinct colour bands within a single character cell. A character that would normally be limited to ink-on-paper suddenly has up to eight colours (two per band, four bands). In practice, the effect is striking — sprites and tiles display far more colour detail than the Spectrum's architecture was designed to allow.

Double buffering

GLUF maintains two complete sets of LDPUSH display code. While one set executes (drawing the current frame), the other is being patched with new tile and sprite data for the next frame. This eliminates the flicker that would result from modifying display code while it is executing.

The cost is memory. Each set of display code covers the 24x16 character game area at 51 bytes per scanline times 128 scanlines: roughly 6,500 bytes per buffer. Two buffers consume about 13,000 bytes. On a 128K Spectrum with banked memory, this is manageable but significant — it means careful memory planning for the rest of the game's assets.

The two-frame architecture

Here is where the engineering gets brutal. GLUF does not render a complete frame every 1/50th of a second. It uses a two-frame architecture:

Frame 1: Change attributes for the multicolor effect, then render as many tiles as possible into the display code buffer. The attribute changes are the time-critical part — they must happen at precisely the right moment relative to the raster beam.

Frame 2: Finish rendering any remaining tiles, then overlay sprites onto the display code buffer.

The split is necessary because the workload simply does not fit in one frame. Rendering tiles into the LDPUSH buffer means patching operand bytes inside the display code — for each tile pixel, you calculate which LD instruction it affects and write the new value into the right byte offset. This is not a simple block copy. The interleaved structure of the display code (opcode-data-data-opcode-data-data…) means tile rendering involves scattered writes, not sequential ones.

The total rendering budget is approximately 70,000 T-states per frame — nearly the entire Pentagon budget of 71,680. What remains is barely enough for game logic, input handling, and the periodic sound update.

Sound runs at 25 Hz instead of 50 Hz. Every other frame, the engine skips the sound update entirely to reclaim those T-states for rendering. The player does not notice the halved update rate for simple sound effects. For music, the 25 Hz rate means each note lasts twice as many frames, which requires the music engine to be written specifically for this constraint.

What the player sees

The player does not see any of this. They see a side-scrolling game with more colour than a Spectrum should have. Sprites move over scenery without the usual attribute clash. Tiles display shading, texture, and multi-coloured detail that would be impossible with standard 8x8 attributes. The game feels like it belongs on a more capable platform.

This is the point DenisGrachev's work makes forcefully: multicolor is not a demo trick. It is a game engine technique. The engineering is extreme — double buffers, two-frame rendering, 25 Hz sound — but the result is a playable game with visuals that genuinely break the Spectrum's perceived limits.

Ringo: A Different Kind of Multicolor

In December 2022, DenisGrachev published a second article on Hype: "Ringo Render 64x48." Where GLUF extended the attribute grid to 8x2, Ringo threw it away entirely and built something closer to a chunky-pixel framebuffer with per-pixel colour.

The approach is conceptually simple and technically devious.

The 11110000b pattern

Fill every pixel byte in screen memory with the value `$F0` — binary `11110000`. The left four pixels of each byte are set (ink colour), and the right four are clear (paper colour). Now, if you change the attribute for that cell, the left half displays the ink colour and the right half displays the paper colour. A single 8-pixel-wide cell displays two distinct colours, side by side.

With standard 8x8 attributes, this gives you a 64x24 grid of independently coloured "pixels," each 4 real pixels wide and 8 real pixels tall. Not bad, but not revolutionary.

Two-screen switching

The trick that makes Ringo work: the ZX Spectrum 128K has two screen buffers. Screen 0 lives at `$4000`, Screen 1 at `$C000` (in bank 7). A single `OUT` to port `$7FFD` switches which screen the ULA displays.

Ringo prepares both screens with the `11110000b` pattern but with *different* attributes on each screen. It then switches between the two screens every 4 scanlines. The effect: within each 8-pixel-tall character row, the top 4 scanlines show Screen 0's attributes, and the bottom 4 show Screen 1's. Each half can specify independent ink and paper colours.

Combined with the `11110000b` pixel pattern, this gives:

- 2 colour columns per character cell (left 4 pixels = ink, right 4 = paper)
- 2 colour rows per character cell (top 4 scanlines from Screen 0, bottom 4 from Screen 1)
- Total: 4 independently coloured sub-cells per 8x8 character cell

Over the full screen: 64 columns x 48 rows = **3,072 independently coloured pixels**, each 4x4 real pixels in size. The effective resolution is 64x48 with full per-pixel colour from the Spectrum's 15-colour palette.

This is a fundamentally different approach from GLUF's 8x2 multicolor. GLUF changes attributes in sync with the beam, requiring precise timing and consuming massive cycle budgets. Ringo uses dual-screen hardware switching, which requires only a single `OUT` instruction every 4 scanlines. The CPU overhead for the screen switching itself is minimal.

Where the T-states go

The cheap screen switching means more T-states are available for game logic. But rendering into two screens simultaneously is not free. Every tile and sprite update must be written to both Screen 0 and Screen 1, because the player sees a composite of both.

Ringo's sprites are 12x10 "pixels" in the 64x48 grid, which means 12 attribute bytes wide and 10 attribute rows tall (split across the two screens). Each sprite occupies 120 bytes of data. Sprite rendering uses fixed-cycle macros — sequences of instructions with known, constant execution time, critical for maintaining synchronisation with the screen switching.

Tile rendering is more involved. DenisGrachev pre-generates tile rendering code in memory pages, using `pop af : or (hl)` patterns:

```
z80 id:ch08_where_the_t_states_go ; Tile rendering fragment (conceptual)     pop
af             ; 10 T  load tile data from stack-based source     or   (hl)
;  7 T  combine with existing screen data     ld   (hl),a         ;  7 T  write
back     inc  l              ;  4 T  next attribute column                          ;
---                         ; 28 T per attribute byte
```

The `pop af` is a stack trick: the tile data is arranged as a stack-format table in memory. SP points to the tile data, and POP reads two bytes at a time. The `or (hl)` combines the tile colour with whatever is already on screen, allowing transparent tiles and layered backgrounds.

Horizontal scrolling

Ringo implements horizontal scrolling with half-character displacement. Since each "pixel" in the 64x48 grid is 4 real pixels wide, scrolling by one "pixel" means shifting the `11110000b` pattern by 4 bits. But the pattern is fixed — you cannot easily shift it without corrupting the colour trick.

Instead, DenisGrachev scrolls by moving the attribute data. A one-pixel scroll shifts all attributes one column to the left or right and draws the new column at the edge. Because the attributes are the only thing that changes (the pixel pattern stays fixed at `$F0`), the scroll is just a block copy of attribute bytes. For the 64x48 grid, this is 48 bytes per column shift (one byte per row), far cheaper than pixel-level scrolling.

For sub-"pixel" scrolling — smooth movement within a 4-pixel-wide column — Denis-Grachev alternates between `$F0` and `$E0` (or similar shifted patterns) in the pixel data. This requires more bookkeeping but achieves half-character displacement, giving the illusion of 128-column horizontal resolution.

Traditional Multicolor: The Interrupt-Driven Approach

Before we move to the practical, it is worth understanding the "classic" approach that GLUF and Ringo displaced. Traditional multicolor is conceptually the simplest: change attributes at precisely the right moment, and the ULA will display different colours on different scanlines.

The technique works like this:

1. Execute `HALT` to synchronise with the frame interrupt. After HALT, the CPU is at a known T-state position relative to the start of the display.
2. Count T-states from the HALT. The ULA reads each row of 32 attributes at a known point in each scanline. By padding with `NOP`s or other known-length instructions, you can position your code at exactly the right moment.
3. At the precise T-state when the ULA has finished reading attributes for the current scanline (but before it reads them for the next), overwrite the 32 attribute bytes with new values.
4. Wait for the ULA to read the new values, then overwrite again for the next change.

The timing is brutal. Each scanline takes 224 T-states on Pentagon. The ULA reads 32 attribute bytes early in each scanline, and the CPU must change all 32 bytes in the gap before the next read. With `LD (HL),A : INC L` at 11 T-states per byte, writing 32 bytes takes 352 T-states — more than one entire scanline. You cannot change every scanline. At best, you can change every other scanline (8x2 resolution) if you use the fastest possible output method (PUSH-based), and even then the timing margins are razor-thin.

"'mermaid id:ch08_traditional_multicolor_the sequenceDiagram participant CPU participant ULA as ULA (electron beam) participant ATTR as Attribute RAM ($5800)

```
Note over ULA: Scanline N begins (224T)
ULA->>ATTR: Read 32 attribute bytes for row
```

```
Note over ULA: Display scanline N using attrs

rect rgb(200, 230, 255)
    Note over CPU: ← Window: CPU rewrites attributes
    CPU->>ATTR: Write 32 new attribute bytes
    Note over CPU: (must complete before next read!)
end

Note over ULA: Scanline N+2 begins
ULA->>ATTR: Read 32 attribute bytes (sees NEW values)
Note over ULA: Display scanline N+2 with new colours

rect rgb(200, 230, 255)
    Note over CPU: ← Window: CPU rewrites again
    CPU->>ATTR: Write 32 more attribute bytes
end

Note over ULA: Scanline N+4 begins
ULA->>ATTR: Read attrs (sees NEWEST values)
Note over ULA: Display with third colour band
```

> **The race:** At 224 T-states per scanline, writing 32 bytes via `LD (HL),A : INC L` costs 3
based attribute rewriting, and why LDPUSH merges pixel and attribute output to avoid a separat

The practical result: traditional multicolor consumes 80--90% of the CPU on attribute manageme
states remain for game logic, collision detection, or sound.

DenisGrachev's LDPUSH technique solves this by merging the attribute output with the pixel out

---

Sidebar: Black Crow #05 --- Early Multicolor

The technique of changing attributes between scanlines was documented as early as 2001 in Blac
states, change attributes --- along with working example code.

Black Crow is significant as a historical marker. By 2001, the ZX Spectrum demoscene had evolv

DenisGrachev's contribution, nearly two decades later, was not inventing multicolor but solvin

---

The Spectrum's Palette and Multicolor

A brief note on colour mechanics, because multicolor's visual impact depends entirely on the p

The ZX Spectrum has 15 colours: 8 base colours (black, blue, red, magenta, green, cyan, yellow

With 8x2 multicolor, each character cell gets four attribute rows instead of one. Each row spe

With Ringo's 64x48 approach, each sub-cell is fully independent. The 15-colour palette is avai

```
bit home computers with more capable hardware --- the MSX2, the Amstrad CPC --- could achieve

---

## Practical: A Multicolor Game Screen

Let us build a simplified multicolor renderer. The goal: a 24-character-wide game area with 8x
frame rendering approach.

### Step 1: The display code buffer

First, we need a block of memory filled with LDPUSH instruction pairs. For each scanline in th
character game area, we need 12 pairs of `LD DE,nn : PUSH DE` (each pair outputs 2 bytes, 12 p

```z80 id:ch08_step_1_the_display_code
; Structure of one scanline's display code (conceptual)
; SP is pre-set to the right edge of this scanline in screen memory

    ld   de,$0000       ; 10 T  rightmost 2 bytes (will be patched)
    push de             ; 11 T
    ld   de,$0000       ; 10 T  next 2 bytes leftward
    push de             ; 11 T
    ld   de,$0000       ; 10 T
    push de             ; 11 T
    ; ... 12 pairs total ...
    ld   de,$0000       ; 10 T  leftmost 2 bytes
    push de             ; 11 T
    ; --- 252 T per scanline (12 x 21) ---

    ; Then: adjust SP for the next scanline
    ; Then: change attributes (every 2nd scanline)
```

The total display code for 128 scanlines (16 character rows x 8 scanlines each) at approximately 51 bytes per scanline is about 6,500 bytes.

Step 2: Attribute changes within the display code

Every two scanlines, the display code must include attribute writes. Between the LDPUSH sequences for scanlines N and N+2, insert code that overwrites the 32 attribute bytes for the current character row:

“‘z80 id:ch08_step_2_attribute_changes ; After outputting scanline N... ; Attribute change for the next 2-scanline band

```
ld   sp,attr_row_end       ; point SP at end of attribute row
ld   de,attr_data_0        ; 10 T  rightmost 2 attribute bytes
push de                    ; 11 T
ld   de,attr_data_1        ; 10 T
push de                    ; 11 T
; ... 16 pairs for 32 attribute bytes ...

ld   sp,next_scanline_end  ; point SP at next scanline's right edge
; Continue with pixel LDPUSH pairs for scanlines N+2, N+3
```
```

```
The attribute changes are embedded directly in the display code stream. They execute at exactl
state counting required. No NOP padding. The structure of the code guarantees the timing.

### Step 3: Tile rendering via operand patching

To draw a tile into the game area, you must patch the operand bytes of the LD instructions in

```z80 id:ch08_step_3_tile_rendering_via
; Patch one scanline of a 16x16 tile into the display buffer
; IX points to the LD DE instruction for this position in the buffer
; HL points to the tile's pixel data for this scanline

    ld   a,(hl)             ;  7 T  read tile byte 0
    ld   (ix+1),a           ; 19 T  patch into LD DE operand (low byte)
    inc  hl                 ;  6 T
    ld   a,(hl)             ;  7 T  read tile byte 1
    ld   (ix+2),a           ; 19 T  patch into LD DE operand (high byte)
    inc  hl                 ;  6 T
    ; advance IX to the next scanline's LD DE instruction
    ; (stride depends on display code structure)
```

At 19 T-states per IX-indexed write, this is not cheap. For a full 16x16 tile (16 scanlines x 2 bytes x 2 patches per byte): roughly 1,200 T-states per tile. In a 24x16 character game area with 2-character-wide tiles, there are up to 192 tiles. Even with partial updates (only redrawing tiles that changed), tile rendering dominates the budget.

This is why GLUF splits rendering across two frames. Frame 1 handles the time-critical attribute changes and renders as many tiles as possible. Frame 2 finishes tiles and composites sprites on top.

Step 4: Sprite overlay

Sprites are rendered on top of tiles using the same operand-patching technique, but with an additional step: save the original operand bytes before overwriting them, so the sprite can be erased on the next frame by restoring the saved data.

```
z80 id:ch08_step_4_sprite_overlay ; Sprite rendering: save background, patch
sprite data     ld   a,(ix+1)           ; read current (background) byte     ld
(save_buffer),a    ; save for later restoration     ld   a,(sprite_data)    ;
load sprite pixel     ld   (ix+1),a           ; patch into display code
```

The save/restore mechanism is the multicolor equivalent of dirty-rectangle sprite rendering. You save what was there, draw the sprite, display it, then restore the saved bytes to erase the sprite before drawing it in its new position.

Step 5: The main loop

“‘z80 id:ch08_step_5_the_main_loop main_loop: halt ; synchronise with frame

```
; --- Frame 1: attributes + tiles ---
di
ld   (restore_sp+1),sp  ; save real SP
```
```

```
call execute_display     ; run the LDPUSH display code
```

restore_sp: ld sp,$0000 ; restore SP ei

```
call update_tiles        ; patch changed tiles into buffer B
call read_input          ; handle player input
call update_game_logic   ; move entities, check collisions

halt                    ; synchronise with next frame

; --- Frame 2: remaining tiles + sprites ---
di
ld   (restore_sp2+1),sp
call execute_display     ; display the current frame
```

restore_sp2: ld sp,$0000 ei

```
call finish_tiles        ; patch any remaining tiles
call erase_old_sprite    ; restore saved bytes
call render_sprite       ; patch sprite into new position
call update_sound        ; sound at 25 Hz (every other frame pair)

jr   main_loop


This skeleton captures the essential rhythm: two frames per logical game frame, display code e
frame structure is the engineering heart of a multicolor game engine.

---

## What DenisGrachev's Work Means

DenisGrachev's achievement was not inventing multicolor --- the technique was known. It was so
frame architecture, the merged code-as-data display buffer, and the 25 Hz sound compromise are
screen switching.

---

## Summary

- **Attribute clash** (two colours per 8x8 cell) is the Spectrum's defining visual constraint.
- The **LDPUSH technique** fuses display data with executable code: `LD DE,nn : PUSH DE` seque
- **GLUF** achieves 8x2 multicolor in a 24x16 character game area with double-
buffered display code, 16x16 sprites and tiles, and a two-frame architecture that splits rende
- **Ringo** uses the `11110000b` pixel pattern with dual-screen switching every 4 scanlines to
pixel colour --- a fundamentally different trade-off that favours colour independence over spa
- **Traditional multicolor** (interrupt-driven attribute changes) is conceptually simpler but
90% of the CPU, making it impractical for games.
- The **Black Crow #05** magazine documented multicolor as early as 2001. DenisGrachev's contr
- DenisGrachev's work demonstrates that demoscene techniques are engineering tools, not just d

---

## Try It Yourself
```

1. **Build the display buffer.** Write a program that generates a block of `LD DE,nn : PUSH DE
execute to see the screen update.

2. **Add attribute changes.** Extend your display buffer to include attribute writes every 2 s

3. **Patch a tile.** Write a routine that takes a 16x16 pixel tile and patches it into the dis

4. **Move a sprite.** Implement save/restore background patching: before drawing the sprite, s

5. **Measure the budget.** Use the border-colour timing harness from Chapter 1 to measure how
states. See how much room remains for game logic.

> **Sources:** DenisGrachev, "Multicolor Will Be Conquered" (Hype, 2019); DenisGrachev, "Ringo

\newpage

Chapter 9: Attribute Tunnels and Chaos Zoomers

> *"This is a VERY BUGGY demo. It is THE hardest thing I've ever done in a demo -- no joke."*
> -- Introspec, file_id.diz for the party version of Eager (to live), 3BM Open Air 2015

In the summer of 2015, Introspec sat down to build something he had never attempted before. Th

This chapter is a making-of. We will work through the two core visual effects from Eager -- th

---

The Attribute Grid as a Framebuffer

Every ZX Spectrum coder knows the attribute area at `$5800`--`$5AFF`. Each of the 768 bytes co

The insight is disarmingly simple. If you fill pixel memory with a fixed pattern -- say, alter

At 32x24, the resolution is terrible by any normal standard. But Introspec was not building a

Think about what a tunnel looks like from the viewer's perspective. The "mouth" of the tunnel
resolution display: coarse resolution in the centre (where the tunnel is far away and detail d

Introspec pushed this further with pseudo-chunky rendering. In the centre of the screen, sever

This is the first lesson of Eager: the attribute grid is not a limitation to work around. It i

---

Plasma: The Colour Engine

```
The tunnel's colours do not come from a stored texture mapped onto a tube. They come from a pl
of-sines approach that has been a demoscene staple since the Amiga days, here adapted for the

The basic idea: for each position on the 32x24 grid, sum several sine waves with different fre

On the Z80, this means table lookups. A 256-byte sine table, page-aligned so you can index it

The tunnel shape is implicit, not explicit. There is no distance-from-centre calculation, no a

<!-- figure: ch09_tunnel_plasma_computation -->
```mermaid id:ch09_plasma_the_colour_engine
graph TD
    A["For each attribute cell (row, col)"] --> B["Look up sin(col × freq1 + phase1)<br>from 2
byte sine table"]
    B --> C["Look up sin(row × freq2 + phase2)"]
    C --> D["Look up sin(col+row + phase3)"]
    D --> E["Sum the three sine values"]
    E --> F["Index into colour map<br>(sum → attribute byte)"]
    F --> G["Write attribute to buffer"]
    G --> H{More cells<br>in quadrant?}
    H -- Yes --> A
    H -- No --> I["4-fold symmetry copy:<br>top-left → top-right,<br>bottom-
left, bottom-right"]
    I --> J["Increment phase1, phase2, phase3<br>→ next frame"]
    J --> A

    style F fill:#f9f,stroke:#333
    style I fill:#bfb,stroke:#333
```

> **Key insight:** There is no distance-from-centre calculation, no angle table, no polar coordinate transform. The tunnel shape emerges from sine wave interference — concentric rings appear naturally from overlapping frequencies. Only one quarter (16×12) is computed; the rest is mirrored.

This is cheaper than a true geometric tunnel (which would require per-pixel distance and angle lookups) and produces a visually rich result. The trade-off is less geometric precision, but at 32x24 resolution, geometric precision was never on the table anyway.

Four-Fold Symmetry: Divide and Conquer

Even at 32x24, calculating plasma for all 768 cells every frame is expensive on a 3.5MHz Z80. Introspec cut the workload by a factor of four with a classic optimisation: exploit the tunnel's natural symmetry.

A tunnel viewed head-on is symmetric about both the horizontal and vertical axes. If you calculate one quarter of the screen – the top-left 16x12 block – you can copy it to the other three quarters by mirroring. Top-left to top-right is a horizontal flip. Top-left to bottom-left is a vertical flip. Top-left to bottom-right is both.
```

The copy routine is tight. In Introspec's implementation, HL points to the source byte in the top-left quarter, and the three destination addresses (top-right, bottom-left, bottom-right) are maintained in a combination of absolute addresses and register pair BC:

```
z80 id:ch09_four_fold_symmetry_divide_and     ld a,(hl)      ; read source
byte from top-left quarter     ld (nn),a      ; write to upper-right quarter
(mirrored)     ld (mm),a      ; write to lower-left quarter (mirrored)     ld
(bc),a      ; write to lower-right quarter (mirrored)     ldi            ; copy
source to its own destination AND advance HL, DE
```

The `ld (nn),a` and `ld (mm),a` instructions use absolute addressing – the target addresses are embedded directly in the code, patched via self-modification or code generation for each cell position. The `ldi` instruction at the end does double duty: it copies the byte from (HL) to (DE) for the top-left quarter's own position in the attribute buffer, and it auto-increments both HL and DE while decrementing BC. This means the loop counter, the source pointer advance, and one of the four writes are all folded into a single two-byte instruction.

The total cost: under 15 T-states per byte for the four-way copy. For 192 source bytes (one quarter of the 768-byte attribute area), that is roughly 2,880 T-states to fill the entire screen. At 3.5MHz with a ~70,000 T-state frame budget, this leaves the vast majority of the frame for the plasma calculation, the music engine, and the digital drum playback that made Eager distinctive.

The addresses `(nn)` and `(mm)` are literal two-byte values baked into `LD (addr),A` instructions, patched via self-modification or code generation for each cell position. This is standard demoscene practice: the Z80's lack of an instruction cache means self-modifying code executes reliably.

The Chaos Zoomer

The second major visual effect in Eager is the chaos zoomer. Where the tunnel is smooth and organic, the zoomer is jagged and fractal-like – a field of attribute data zooming toward or away from the viewer, with new detail emerging at the edges as the zoom progresses.

The "chaos" comes from the visual result, not the algorithm. The effect zooms into a region of attribute data, magnifying the centre while the edges scroll inward. Because the source data has patterns at multiple scales, the zooming reveals self-similar structure that reads as fractal to the eye.

The implementation relies on unrolled `ld hl,nn : ldi` sequences. Each `ld hl,nn` loads a new source address – the position in the source buffer to sample for this particular output cell. The following `ldi` copies from (HL) to (DE), advancing DE to the next output position. The source addresses are arranged so that cells near the centre of the screen sample from nearby positions in the source data (magnification), while cells near the edges sample from widely spaced positions (compression). Vary the mapping over time and the zoom animates.

```
z80 id:ch09_the_chaos_zoomer     ; Unrolled chaos zoomer fragment     ld hl,src_addr_0
; source for output cell 0     ldi                 ; copy to output, advance DE
```

```
ld hl,src_addr_1    ; source for output cell 1     ldi     ld hl,src_addr_2    ;
source for output cell 2     ldi     ; ... repeated for all 768 cells (or one
quarter, with symmetry)
```

The key optimisation: since `ldi` auto-increments DE, you never need to calculate or load the destination address. The output is always written sequentially into attribute RAM. Only the source addresses vary, and they are embedded directly in the instruction stream as immediate operands. This makes the zoomer a long sequence of `ld hl,nn : ldi` pairs – conceptually simple, but each pair is just 5 bytes (3 for `ld hl,nn` + 2 for `ldi`) and 26 T-states. For a full quarter-screen of 192 cells, that is roughly 5,000 T-states of pure copying, plus the four-way symmetry copy on top.

The complication is that the source addresses change every frame as the zoom progresses. Updating 192 two-byte addresses embedded in the code would cost nearly as much as the copy itself. This is where code generation enters the picture.

Code Generation: Processing Writes Z80

Introspec did not write the zoomer's unrolled code by hand. The address sequences are different for every zoom level, and calculating them at runtime would consume the frame budget. Instead, he wrote the code generator in Processing, the Java-based creative coding environment. A Processing sketch calculated, for each frame and each output cell, which source cell should be sampled, then output a complete `.a80` source file containing the unrolled `ld hl,nn : ldi` sequence with all addresses filled in. sjasmplus compiled this generated source alongside the hand-written engine code.

The pipeline: Processing calculates the zoom mapping, writes `.a80` source, the assembler compiles it, and at runtime the scripting engine selects which pre-generated frame to execute. The Z80 does not compute the mapping. It merely plays it back.

This trades memory for speed – the pre-generated code for all zoom frames occupies substantial RAM, hence the 128K requirement – but the runtime cost per frame is minimal.

The Zapilator Question

The ZX scene has a long and occasionally heated relationship with precalculation. The Russian demoscene coined the term *zapilator* – roughly, "precalculator" – for demos that rely heavily on pre-generated data rather than real-time computation. The word carries a faint whiff of disapproval. If the PC does all the interesting work, what is the Spectrum actually doing? Is it a demo or a slideshow?

Introspec's answer is characteristically nuanced. The art, he argues, is not in the computation itself but in *designing what to precalculate*. Choosing the right zoom mapping, the right interpolation, the right way to decompose the problem so that the pre-generated code fits in memory and the playback runs at 50Hz – this is

engineering. The Processing script does not write itself. The Z80 code structure that makes playback efficient does not emerge automatically. The creativity lives in the architecture, not in whether the inner loop contains an `add` or a `ldi`.

He has a point. The chaos zoomer's visual quality depends on the source data, the mapping function, the zoom curve, the colour palette, and the interplay with the music. All of these are artistic decisions. The fact that the address calculations happen at compile time rather than runtime is an implementation detail – one that enables a visual quality impossible with real-time computation at 3.5MHz. The machine's constraints – its memory, its instruction set, its timing – shaped every decision. That the shaping happened partly in Processing and partly in Z80 assembly does not diminish the result.

For this book's purposes, the takeaway is practical: code generation is a legitimate and powerful technique. If your effect requires calculations that exceed the Z80's frame budget, consider moving them to build time. Your assembler's macro language, a Lua script inside sjasmplus, or an external program in Python or Processing can all serve as code generators. The Z80 gets to do what it does best: copy data at maximum speed.

A Taste of the Scripting Engine

Eager contains more than the tunnel and the zoomer. It runs for two minutes, with multiple visual variations, transitions, and the digital drum playback that gives the demo its rhythmic pulse. Coordinating all of this is a scripting engine – a topic we will explore in depth in Chapter 12 when we discuss music synchronisation. But a brief sketch here sets up that discussion.

Introspec's engine uses two levels of scripting. The **outer script** controls the sequence of effects: play the tunnel for N frames, transition to the zoomer, back to the tunnel with different parameters, and so on. The **inner script** controls variations within a single effect: changing the plasma frequencies, shifting the colour palette, adjusting the zoom speed.

A critical command in the scripting language is what Introspec calls **kWORK**: "generate N frames, then show them independently." This is the key to Eager's asynchronous frame generation. The engine pre-renders several frames of the current effect into memory buffers. Then, while those frames are being displayed (one per screen refresh), the engine can do other work – like playing a digital drum sample through the AY chip.

This async architecture is what made Eager so difficult to build. When a drum hit occurs, the CPU is consumed by digital sample playback. Frame generation stalls. The visual effect survives on pre-rendered frames until the drum finishes and generation resumes. During rapid drum hits, the generator falls behind; between hits, it catches up. "My brain is not coping with asynchronous coding well," Introspec wrote in the party version's file_id.diz. The honest exhaustion in that note captures the reality of interleaving a timing-critical audio routine with a frame generation pipeline on a machine with one thread and no operating system.

We will return to this architecture in Chapter 12, where we also examine n1k-o's digital drum technique and the double-buffered attribute frames that make the

whole system work.

The Making Of: Timeline and Inspiration

Eager was developed between June and August 2015. Introspec has said the initial inspiration came from seeing the twister effect in **Bomb** by Atebit – a visual trick that exploited attribute manipulation to create the illusion of a three-dimensional rotating column. "What else could you do with attributes alone?" was the question that started the project.

The music came from n1k-o (of Skrju), whose track gave the demo its rhythmic structure. The hybrid drum technique – digital sample for the attack transient, AY envelope for the decay – was n1k-o's innovation, and it drove the entire architectural decision to build an asynchronous frame generation engine. Without the drums, Eager could have been a simpler demo. With them, it became what Introspec called "the hardest thing I've done in a demo."

The development was compressed into roughly ten weeks. The party version, submitted to 3BM Open Air 2015, still had bugs – the file_id.diz carried a note thanking diver of 4th Dimension "for the cool tip" and apologising for the instability. The final version fixed the timing issues across Spectrum models (128K, +2, +2A/B, +3, Pentagon – all at 3.5MHz only, no turbo). That cross-pollination – a scener from one group passing a technical insight to a coder from another – is how the ZX demoscene evolves.

Practical: Building a Simplified Attribute Tunnel

Let us build a stripped-down version of Eager's attribute tunnel. We will implement:

1. A fixed pixel pattern in bitmap memory (so attributes have something to colour).
2. A plasma calculation over the 32x24 attribute grid.
3. Four-fold symmetry to reduce the calculation to one quarter.
4. Animation by incrementing the plasma phase each frame.

This will not match Eager's visual sophistication – we are omitting the pseudo-chunky variable-size pixels, the code-generated zoomer, and the async architecture. But it will demonstrate the core principle: the attribute grid as your framebuffer.

Step 1: Fill Pixel Memory with a Pattern

We need a fixed pixel pattern so that ink and paper colours are both visible. A simple checkerboard works:

```
z80 id:ch09_step_1_fill_pixel_memory_with ; Fill bitmap memory ($4000-$57FF)
with checkerboard pattern     ld hl,$4000     ld de,$4001     ld bc,$17FF          ;
6143 bytes     ld a,$55             ; 01010101 binary -- alternating pixels
ld (hl),a     ldir
```

Every 8x8 cell will now display alternating ink and paper pixels. When we change the attribute, the checkerboard reveals both colours.

Step 2: Sine Table

Page-align a 256-byte sine table for fast indexing. This can be generated at assembly time using sjasmplus's Lua scripting, or pre-calculated and included as binary data:

```
lua id:ch09_step_2_sine_table     ALIGN 256 sin_table:     LUA ALLPASS     for i
= 0, 255 do         -- Sine scaled to 0..63 (6-bit unsigned)         sj.add_byte(math.floor(ma
* math.pi / 128) * 31 + 32))     end     ENDLUA
```

Step 3: Plasma for One Quarter

Calculate the plasma value for each cell in the top-left 16x12 quarter. The result is an index into a colour table that produces the attribute byte:

"'z80 id:ch09_step_3_plasma_for_one_quarter ; Calculate plasma for top-left quarter (16 columns x 12 rows) ; Input: frame_phase is incremented each frame ; Output: attr_buffer filled with 192 attribute bytes

calc_plasma: ld iy,attr_buffer ld h,sin_table / 256 ; H = high byte of sine table page ld b,12 ; 12 rows (half of 24) .row_loop: ld c,16 ; 16 columns (half of 32) .col_loop: ; Plasma = sin(x*2 + phase1) + sin(y*3 + phase2) + sin(x+y + phase3)

```
; Term 1: sin(x*2 + phase1)
ld a,c
add a,a                  ; x * 2
add a,(ix+0)             ; + phase1 (self-modifying or IX-indexed)
ld l,a
ld a,(hl)                ; sin_table[x*2 + phase1]
ld d,a                   ; accumulate in D

; Term 2: sin(y*3 + phase2)
ld a,b
add a,a
add a,b                  ; y * 3
add a,(ix+1)             ; + phase2
ld l,a
ld a,(hl)                ; sin_table[y*3 + phase2]
add a,d
ld d,a

; Term 3: sin((x+y) + phase3)
ld a,c
add a,b                  ; x + y
add a,(ix+2)             ; + phase3
ld l,a
ld a,(hl)                ; sin_table[x+y + phase3]
add a,d                  ; total plasma value

; Map to attribute byte
rrca
```

```
rrca
rrca                    ; shift to get ink bits in position
and %00000111           ; 8 ink colours
or  %00111000           ; white paper (bits 3-5 = 7)
ld (iy+0),a             ; store in buffer
inc iy

dec c
jr nz,.col_loop
djnz .row_loop
ret


This is intentionally simplified. A production version would use self-modifying code to inline
states per load versus 19 for IX-indexed) and a 256-byte colour lookup table instead of the `r

### Step 4: Four-Way Copy

The simplified approach copies row by row, mirroring horizontally for the right half and index
pass copy we saw earlier -- it writes all four quadrants simultaneously from one source byte,
modifying `ld (nn),a` instructions with pre-patched addresses. The practical code for that pat

### Step 5: Main Loop

```z80 id:ch09_step_5_main_loop
main_loop:
    halt                    ; wait for vsync

    call calc_plasma        ; calculate one quarter
    call copy_four_way      ; mirror to full screen

    ; Advance plasma phases
    ld hl,phase1
    inc (hl)
    inc (hl)                ; phase1 += 2
    ld hl,phase2
    inc (hl)                ; phase2 += 1
    ld hl,phase3
    dec (hl)                ; phase3 -= 1 (counter-rotating)

    jr main_loop
```

The different phase increments make the plasma terms rotate at different speeds. Experiment with these values – even small changes produce dramatically different visual textures.

Key Insight

The attribute grid IS your framebuffer for this effect. You never touch pixel memory after the initial checkerboard fill. The entire animation consists of writing 768 bytes per frame to `$5800–$5AFF`. The Z80's screen address interleaving, which makes pixel manipulation so painful, is completely irrelevant. The attribute area is linear.
```

The copy is fast. The visual result, at 50Hz, is smooth and surprisingly compelling.

This is the lesson Introspec drew from building Eager, and it applies far beyond tunnels. Any time the ZX Spectrum's colour system frustrates you, consider inverting the problem. Instead of fighting the attribute grid, use it. Those 768 bytes are the cheapest full-screen animation buffer on the machine.

Sources

- Introspec, "Making of Eager," Hype, 2015 (hype.retroscene.org/blog/demo/261.html)
- Introspec, file_id.diz from Eager (to live) party version, 3BM Open Air 2015
- Introspec, "Код мёртв" (Code is Dead), Hype, 2015
- Introspec, "За дизайн" (For Design), Hype, 2015

Next: Chapter 10 takes us to Illusion's dotfield scroller and Eager's four-phase colour animation – where two normal frames and two inverted frames create the illusion of a palette the Spectrum does not have.

Chapter 10: The Dotfield Scroller and 4-Phase Colour

> *"Two normal frames and two inverted frames. The eye sees the average."*
> – Introspec, Making of Eager (2015)

The ZX Spectrum displays two colours per 8x8 cell. Text scrolls across a screen at whatever rate the CPU can manage. These are fixed constraints – the hardware does what it does, and no amount of cleverness will change the silicon.

But cleverness can change what the viewer *perceives*.

This chapter brings together two techniques from two different demos, separated by nearly twenty years but connected by a shared principle. The dotfield scroller from X-Trade's *Illusion* (ENLiGHT'96) renders text as a bouncing cloud of individual dots, each placed at a cost of just 36 T-states. The 4-phase colour animation from Introspec's *Eager* (3BM Open Air 2015) alternates four carefully constructed frames at 50Hz to trick the eye into seeing colours the hardware cannot produce. One exploits spatial resolution – placing dots wherever you want, unconstrained by character cells. The other exploits temporal resolution – cycling frames faster than the eye can follow. Together they demonstrate the two main axes of cheating on constrained hardware: space and time.

Part 1: The Dotfield Scroller

What the Viewer Sees

Picture a message – "ILLUSION BY X-TRADE" – rendered not in solid block characters but as a field of individual dots, each dot a single pixel. The text drifts horizontally across the screen in a smooth scroll. But the dots are not sitting on flat scanlines. They bounce. The entire dot field undulates in a sine wave, each column offset vertically from its neighbours, creating the impression of text rippling on the surface of water.

The Font as Texture

The font is stored as a bitmap texture in memory – one bit per dot. If the bit is 1, a dot appears on screen. If the bit is 0, nothing happens. The critical word is *transparent*. In a normal renderer, you write every pixel position. In the dotfield

scroller, transparent pixels are nearly free. You check the bit, and if it is zero, you skip. Only the set pixels require a write to video memory.

This means rendering cost is proportional to the number of visible dots, not the total area. A typical 8x8 character might have 20 set pixels out of 64. For a large scrolling message, this economy matters enormously. BC points to the font data; RLA shifts each bit into the carry flag to determine on or off.

Stack-Based Address Tables

In a conventional scroller, each pixel's screen position is calculated from (x, y) coordinates using the Spectrum's interleaved address formula. That calculation involves shifts, masks, and lookups. Doing it for thousands of pixels per frame would consume the entire budget.

Dark's solution: pre-calculate every screen address and store them as a table that the stack pointer walks through. POP reads 2 bytes and auto-increments SP, all in 10 T-states. Point SP at the table instead of the real stack, and POP becomes the fastest possible address retrieval – no index registers, no pointer arithmetic, no overhead.

Compare POP to the alternatives. `LD A,(HL) : INC HL` fetches one byte in 11 T-states – you would need two such pairs (22 T) to fetch an address, plus `LD L,A / LD H,A` bookkeeping. An indexed load like `LD L,(IX+0) : LD H,(IX+1)` costs 38 T-states for the pair. POP fetches both bytes, increments the pointer, and loads a register pair – 10 T-states, no contest. The price is that you surrender the stack pointer to the renderer. Nothing else can use SP while the inner loop runs.

This means interrupts are fatal. If an interrupt fires while SP points into the address table, the Z80 pushes the return address onto the "stack" – which is actually your data table. Two bytes of carefully computed screen addresses get overwritten with a return address, and the interrupt service routine proceeds to execute whatever garbage sits at the corrupted location. The result is anything from a garbled frame to a hard crash. The solution is simple and non-negotiable: `DI` before hijacking SP, `EI` after restoring it. Every POP-trick routine in every Spectrum demo follows this pattern:

```
z80 id:ch10_stack_based_address_tables     di     ld   (.smc_sp+1), sp  ; save
SP via self-modifying code     ld   sp, table_addr    ; point SP at pre-computed
data     ; ... inner loop using POP ... .smc_sp:     ld   sp, $0000          ;
self-modified: restores original SP     ei
```

The save/restore uses self-modifying code because it is the fastest way to both save and restore SP in one step. `EX (SP),HL` requires a valid stack. `LD (addr),SP` exists (opcode ED 73, 20 T-states), but it saves SP to a fixed address – you would then need a separate `LD SP,(addr)` to restore it later (also 20 T-states), and the restore is no faster than the self-modifying approach. The SMC technique writes SP's value directly into the operand field of a later `LD SP,nnnn` instruction: `LD (.smc+1),SP` costs 20 T-states for the save, and the restore (`LD SP,nnnn` with the patched operand) costs just 10 T-states. The combined save+restore is 30 T-states versus 40 T-states for the LD (addr),SP / LD SP,(addr) pair – a small saving that also avoids reserving a separate memory location.

One subtle consequence: the DI/EI window blocks the frame interrupt. If the inner loop runs long, HALT at the top of the main loop will still catch the next

interrupt – but if the rendering overshoots an entire frame, you lose sync. This is why the budget arithmetic matters. You must know your worst-case timing before committing to the POP trick.

The bouncing motion is encoded entirely in the address table. Each entry is a screen address that already includes the vertical sine offset. The "bounce" does not happen at render time. It happened when the table was built. All three dimensions of the animation – scroll position, bounce wave, character shape – collapse into a single linear sequence of 16-bit addresses, consumed at full speed by POP.

The Inner Loop

Introspec's 2017 analysis of Illusion reveals the inner loop. One byte of font data contains 8 bits – 8 pixels. `LD A,(BC)` reads the byte once, then RLA shifts one bit at a time through 8 unrolled iterations:

"'z80 id:ch10_the_inner_loop ; Dotfield scroller inner loop (unrolled for one font byte) ; BC = pointer to font/texture data, SP = pre-built address table

```
ld   a,(bc)      ;  7 T  read font byte (once per 8 pixels)
inc  bc          ;  6 T  advance to next font byte

; Pixel 7 (MSB)
pop  hl          ; 10 T  get screen address from stack
rla              ;  4 T  shift texture bit into carry
jr   nc,.skip7   ; 12/7 T  skip if transparent
set  7,(hl)      ; 15 T  plot the dot
```

.skip7: ; Pixel 6 pop hl ; 10 T rla ; 4 T jr nc,.skip6 ; 12/7 T set 6,(hl) ; 15 T .skip6: ; ... pixels 5 through 0 follow the same pattern, ; with SET 5 through SET 0 ...

```
The per-pixel cost, excluding the amortised byte-fetch:

| Path | Instructions | T-states |
|------|--------------|----------|
| Opaque pixel | `pop hl` + `rla` + `jr nc` (not taken) + `set ?,(hl)` | **36** |
| Transparent pixel | `pop hl` + `rla` + `jr nc` (taken) | **26** |

The `LD A,(BC)` and `INC BC` cost 13 T-states amortised over 8 pixels -- about 1.6 T per pixel
states per pixel" is the worst-case cost within the unrolled byte, excluding that overhead.

The SET bit position changes for each pixel (7, 6, 5 ... 0), which is why the loop is unrolled
modifying code (overhead). Unrolling is the clean solution.

### The Budget Arithmetic

Let us work the numbers properly. The standard Spectrum 48K frame is 69,888 T-
states (the Pentagon clone runs slightly longer at 71,680). Of that, the ULA steals T-
states during the active display for memory contention, but the scroller writes to screen memo
states on a 48K and 65,000 on a Pentagon. Subtract music playback (a typical AY player costs 3
5,000 T per frame), screen clearing, and table construction. That leaves roughly 40,000-
50,000 T-states for the actual dot rendering.
```

```
Consider a display of 8 characters of 8x8 font = 512 font bits per frame (8 chars x 8 bytes x

- 154 opaque pixels at 36 T each = 5,544 T
- 358 transparent pixels at 26 T each = 9,308 T
- 64 byte-fetches (`LD A,(BC) : INC BC`) at 13 T each = 832 T
- Total: approximately 15,684 T-states

That is well within a single frame. You could render 20+ characters before hitting the budget
150 T-states per entry (depending on implementation), adding 50,000-75,000 T to the frame. Ill
computing the entire table set into memory and cycling through offsets, or by building increme

The numbers work because two optimisations compound. Stack-based addressing eliminates all coo
driven transparency eliminates all writes for empty pixels. The table build is expensive, but
critical DI window and can be spread across the frame.

### How the Bounce Is Encoded

The address table is where the art lives. To create the bouncing motion, a sine table offsets

```text
y_offset = sin_table[(column * phase_freq + scroll_pos * speed_freq) & 255]
```

The two frequency parameters control the visual character of the wave. `phase_freq` determines the spatial frequency – how many wave cycles fit across the visible dot columns. A value of 4 means each dot column advances 4 positions into the sine table, so 256/4 = 64 columns span one full wave cycle. A value of 8 doubles the frequency, creating a tighter ripple. `speed_freq` controls how fast the wave propagates over time: higher values make the bounce scroll faster independently of the text scroll.

The sine table itself is a 256-byte array of signed offsets, page-aligned for fast lookup. Page alignment means the high byte of the table address is fixed; only the low byte changes, so the lookup reduces to:

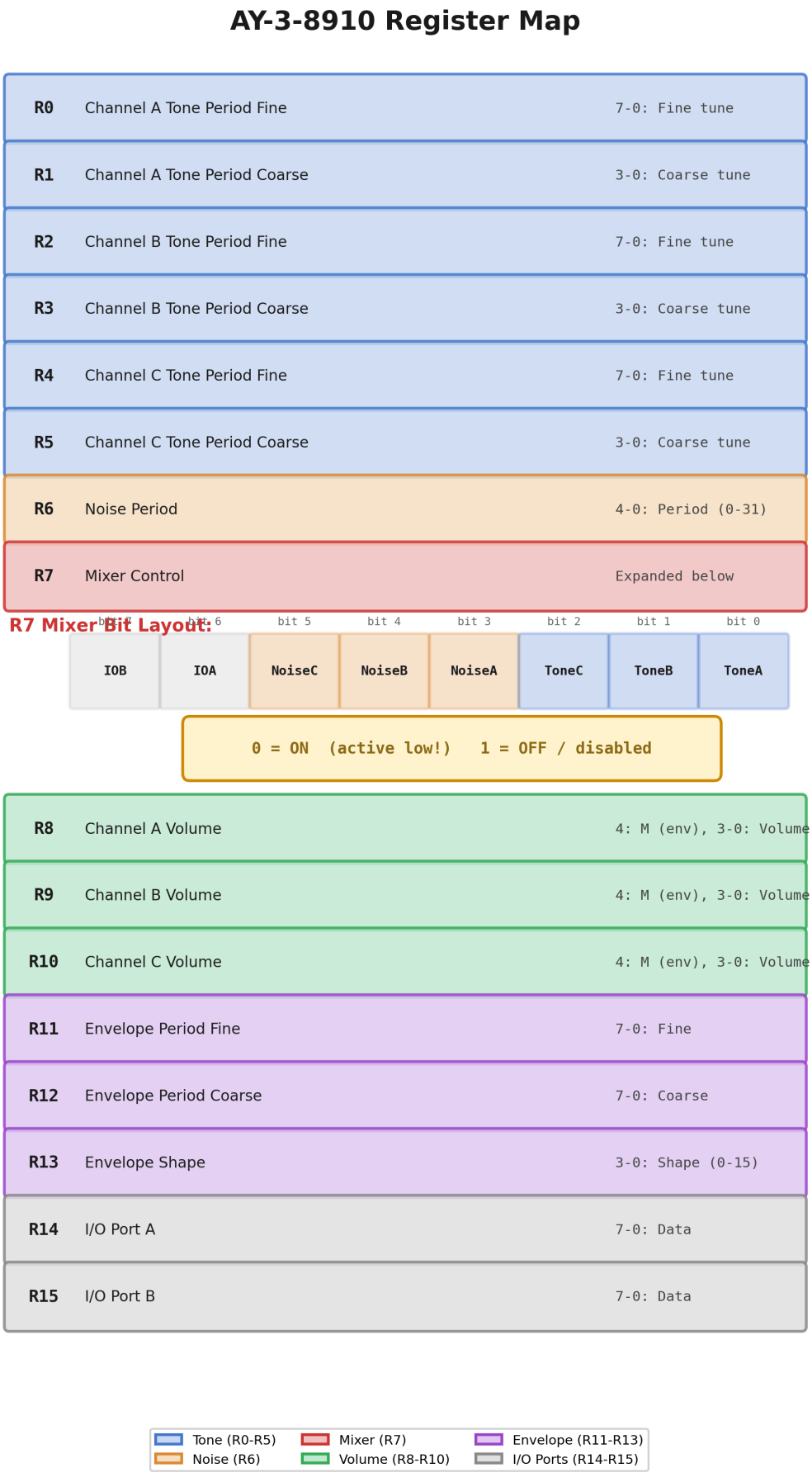
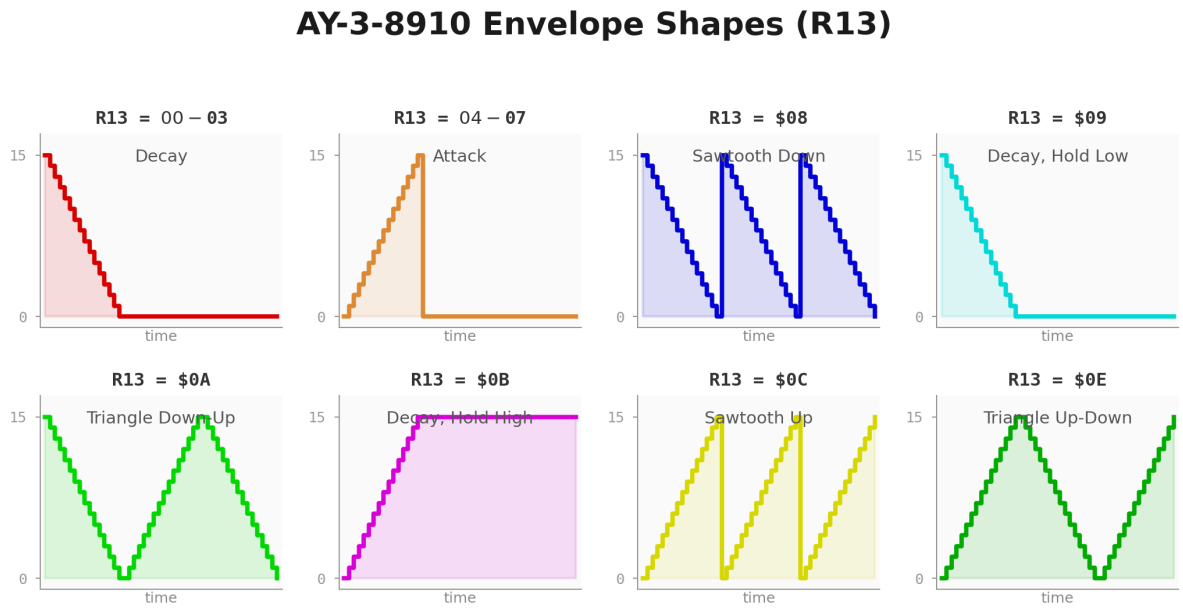
```
z80 id:ch10_how_the_bounce_is_encoded_2     ld   hl, sin_table    ; H = page,
L = don't care     ld   l, a             ; A = (column * freq + phase) & $FF
ld   a, (hl)          ; 7 T — one memory read, no arithmetic
```

The values in the table are signed: positive offsets push the dot down, negative offsets push it up. The amplitude is baked into the table at generation time. A table with range -24 to +24 gives a bounce of 48 scanlines peak-to-peak. Generating the table is a one-time cost, typically done offline or during initialisation using a lookup or a simple approximation. On the Z80, computing true sine values at runtime is expensive, so demoscene coders either pre-compute tables externally or use quadrant symmetry: calculate one quarter-wave (64 entries), then mirror and negate to fill the remaining three quarters.

Given each dot's (x, y + y_offset), the Spectrum screen address is calculated and stored in the table. The table-building code runs once per frame, outside the inner loop. The inner loop sees only a stream of pre-computed addresses.
```

Beyond Simple Sine: Lissajous, Helix, and Multi-Wave Patterns

The beauty of the pre-computed table approach is that the inner loop does not care what shape the motion follows. It consumes addresses at a fixed cost regardless of the trajectory that generated them. This makes it trivial to experiment with different movement patterns – all the complexity lives in the table-building code.

A **Lissajous pattern** adds a horizontal sine offset as well as the vertical one. Instead of each column mapping to a fixed x byte on screen, the x position also oscillates:

```
x_offset = sin_table[(column * x_freq + phase_x) & 255]
y_offset = sin_table[(column * y_freq + phase_y) & 255]
```

When `x_freq` and `y_freq` are coprime (say 3 and 2), the dot field traces a Lissajous figure – the classic oscilloscope pattern. The text becomes a ribbon weaving through space. Different frequency ratios produce dramatically different shapes: 1:1 gives a circle or ellipse, 1:2 gives a figure-eight, 2:3 gives the trefoil pattern familiar from old analogue test equipment.

A **helix** or spiral effect uses a single phase that advances per column, but varies the amplitude:

```
amplitude = base_amp + sin_table[(column * 2 + time) & 255] * depth_scale
y_offset = sin_table[(column * freq + phase) & 255] * amplitude / max_amp
```

This creates the illusion of dots receding into depth – the wave flattens at the "far" point of the spiral and expands at the "near" point.

Multi-wave superposition is the simplest technique with the most dramatic payoff. Add two sine terms with different frequencies:

```
y_offset = sin_table[(col * 4 + phase1) & 255] + sin_table[(col * 7 + phase2) &
↪  255]
```

The result is a complex, organic-looking undulation that never quite repeats. Advancing `phase1` and `phase2` at different speeds produces continuously evolving motion from just two table lookups per column. Three or more harmonics create waves that look almost fluid-dynamic. This is the cheapest possible way to generate complex motion – each additional harmonic costs one table lookup and one addition per column in the table builder, and the inner loop cost remains unchanged.

Part 2: 4-Phase Colour Animation

The Colour Problem

Each 8x8 cell has one ink colour (0-7) and one paper colour (0-7). Within a single frame, you get exactly two colours per cell. But the Spectrum runs at 50 frames per second, and the human eye does not see individual frames at that rate. It sees the average.

The Trick

Introspec's 4-phase technique cycles through four frames:

1. **Normal A:** ink = C1, paper = C2. Pixel data = pattern A.

2. **Normal B:** ink = C3, paper = C4. Pixel data = pattern B.
3. **Inverted A:** ink = C2, paper = C1. Pixel data = pattern A (same pixels, swapped colours).
4. **Inverted B:** ink = C4, paper = C3. Pixel data = pattern B (same pixels, swapped colours).

At 50Hz, each frame displays for 20 milliseconds. The four-frame cycle completes in 80ms – 12.5 cycles per second, above the flicker-fusion threshold on CRT displays.

The Maths of Perception

Trace a single pixel that is "on" in pattern A and "off" in pattern B:

Frame	Pixel state	Displayed colour
Normal A	on (ink)	C1
Normal B	off (paper)	C4
Inverted A	on (ink)	C2
Inverted B	off (paper)	C3

The eye perceives the average: (C1 + C2 + C3 + C4) / 4.

Now check: a pixel "on" in both patterns sees C1, C3, C2, C4. A pixel "off" in both sees C2, C4, C1, C3. All cases produce the same average. The pixel pattern does not affect the perceived hue – only the choice of C1 through C4 does.

Then why have two patterns? Because the *intermediate* transitions matter. A pixel that alternates between bright red and bright green flickers noticeably at 12.5Hz. A pixel alternating between similar shades barely flickers at all. The dithering patterns – checkerboards, halftone grids, ordered matrices – control the *texture* of the flicker. Introspec chose patterns so that transitions between frames produced minimal visible oscillation. This is anti-clashing pixel selection: the careful arrangement of "on" and "off" bits to ensure that no pixel toggles between dramatically different colours in consecutive frames.

Why Inversion Is Essential

Without the inversion step, "on" pixels would always show ink and "off" pixels would always show paper. You would get exactly two visible colours per cell, flickering between two different pairs. The inversion ensures both ink and paper contribute to both pixel states across the cycle, mixing all four colours into the perceived output.

On the Spectrum, inversion is cheap. The attribute byte layout is `FBPPPIII` – Flash, Bright, 3 bits of paper colour, 3 bits of ink colour. Swapping ink and paper means rotating the lower 6 bits: paper moves to ink position, ink moves to paper position, while Flash and Bright stay put. In code:

```
z80 id:ch10_why_inversion_is_essential ; Swap ink and paper in attribute byte
(A) ; Input:  A = F B P2 P1 P0 I2 I1 I0 ; Output: A = F B I2 I1 I0 P2 P1 P0
ld   b, a     and  $C0           ; isolate Flash + Bright bits     ld   c, a
; save FB------     ld   a, b     and  $38           ; isolate paper (--PPP---)
rrca     rrca     rrca              ; paper now in ink position (-----PPP)
```

```
ld   d, a          ; save ink-from-paper     ld   a, b     and  $07          ;
isolate ink (-----III)     rlca     rlca     rlca               ; ink now in
paper position (--III---)     or   d             ; combine: --IIIPPP     or   c
; combine: FBIIIPPP = swapped attribute
```

The alternative is to pre-compute both normal and inverted attribute buffers at initialisation and simply cycle buffer pointers at runtime. This trades 3,072 bytes of memory for zero per-frame computation – a worthwhile trade on 128K machines with memory to spare.

Practical Cost

Four pre-built attribute buffers, cycled once per frame. The per-frame cost is a block copy of 768 bytes into attribute RAM ($5800-$5AFF). Using LDIR, this costs 21 T-states per byte: 768 x 21 = 16,128 T-states. Using the stack trick (POP from the source buffer, switch SP, PUSH to attribute RAM, batching through register pairs and shadow registers), a realistic cost is around 11,000-13,000 T-states depending on batch size and loop overhead – a modest 1.2-1.5x speedup over LDIR. The gain is smaller than you might expect because each batch requires two SP switches (save source position, load destination, then swap back), and that overhead largely offsets the raw speed advantage of POP+PUSH over LDIR. For a *fill* (writing the same value to every byte), the PUSH trick is far more effective – load register pairs once, then PUSH repeatedly – but a copy from varying source data cannot avoid the read cost.

The cycle logic itself is trivial. A single variable holds the phase (0-3). Each frame, increment it and AND with 3 to wrap. Index into a 4-entry table of buffer base addresses:

```
z80 id:ch10_practical_cost     ld   a, (phase)     inc  a     and  3     ld
(phase), a     add  a, a           ; phase * 2 (pointer table is 16-bit entries)
ld   hl, buf_ptrs     ld   e, a     ld   d, 0     add  hl, de     ld   a, (hl)
inc  hl     ld   h, (hl)     ld   l, a           ; HL = source buffer address
ld   de, $5800      ; DE = attribute RAM     ld   bc, 768     ldir               ;
copy attributes for this phase
```

Memory: 4 x 768 = 3,072 bytes for the buffers. On a 48K machine that is a significant chunk; on 128K you can place buffers in paged banks. The pixel patterns (A and B) are written once at initialisation and never touched again – only the attribute RAM changes each frame.

Text Overlay

In Eager, scrolling text overlays the colour animation. There are several approaches, each with different trade-offs.

The simplest is **cell exclusion**: reserve certain character cells for text, skip them during the colour cycle, and write fixed white-on-black attributes with actual font glyphs. This is easy to implement – just mask those cells out of the LDIR copy – but creates a hard visual boundary between the animated background and the static text region. The text looks pasted on.

A more sophisticated approach is **pattern integration**: the glyph shapes override specific bits in both pixel patterns A and B. Where the font has a set bit, both

patterns get that bit set (or cleared, depending on the desired text colour). This ensures the text pixel shows the same colour in all four phases – it does not flicker because it never transitions between different colour states. The surrounding pixels continue to cycle normally. The result is text that appears to float on the animated background, with colour bleeding up to the edges of each letterform. The cost is that you must regenerate (or patch) the pixel patterns whenever the text scrolls, which adds a few thousand T-states per frame depending on how many cells contain text.

A third option for 128K machines is **layer compositing**: maintain the 4-phase background in one set of memory pages and the text scroller in another, then combine them during the attribute copy. This keeps the two systems independent – the scroller does not need to know about the colour animation and vice versa – at the cost of a slightly more complex copy loop that masks text cells.

Demoscene Lineage

The dotfield scroller did not appear from nowhere. The technique sits in a lineage of ZX Spectrum effects that stretches from the mid-1980s to the present.

The earliest Spectrum scrollers were simple character-cell affairs: LDIR-based horizontal scrolls that shifted an entire line of character cells, one byte at a time. Pixel-smooth scrolling was harder – the Spectrum has no hardware scroll register, so every pixel shift requires rewriting the bitmap data. By the early 1990s, demo coders had developed several approaches: RL/RR-based pixel scrolling (shifting every byte in a screen line), look-up table scrollers (pre-shifted copies of each character), and the double-buffer technique (draw into a back buffer, copy to screen). All of these were limited by the fundamental cost of moving bytes in and out of video RAM.

The dotfield approach breaks from this tradition entirely. Instead of scrolling a contiguous block of pixels, it decomposes the text into individual dots and places each one independently. This was Dark's insight in the mid-1990s: if you give up the idea of a solid font and accept a pointillist rendering, you can use the POP trick to place each dot with minimal overhead. The visual result – text dissolving into a cloud of particles, bouncing on a sine wave – became one of the signature effects of the Russian demoscene.

X-Trade's *Illusion* (ENLiGHT'96) was the demo that made the technique famous in the Spectrum world. The dotfield scroller was its centrepiece effect, running smoothly alongside music and other visual elements. Dark published the algorithmic principles in *Spectrum Expert* issues #01 and #02 (1997-98), where he described the general approach to POP-based rendering and sine-table animation. Two decades later, Introspec's detailed reverse-engineering of the Illusion binary (published in *Hype* magazine, 2017) confirmed Dark's claims and provided the exact cycle counts that the community had long speculated about.

The 4-phase colour technique has a different pedigree. Colour-cycling on the Spectrum has been explored since the 1980s – simple two-frame alternation (flash-like effects) was common in games and demos. But the systematic four-phase approach, with its careful inversion step to ensure all four colours contribute equally, was refined by Introspec for *Eager* (3BM Open Air 2015). The party

version's file_id.diz explicitly mentions the technique, and Introspec's "Making of Eager" article in *Hype* (2015) describes the design process: choosing colours so that adjacent phases minimise visible flicker, and using dithering patterns that distribute the transitions evenly across the cell.

The broader principle – temporal multiplexing of colour – appears on other platforms too. The Atari 2600 famously alternates frames to create flickering pseudo-sprites. The Game Boy uses a similar trick for pseudo-transparency. On the Spectrum, the technique is particularly effective because the CRT phosphor persistence smooths the transitions more than an LCD would. This is worth noting for modern viewers: 4-phase colour looks substantially better on a real CRT or a good CRT emulator (with phosphor simulation) than on a raw pixel-perfect display.

The Shared Principle: Temporal Cheating

The dotfield scroller uses 50 frames per second for *spatial* flexibility. Each frame is a snapshot of dot positions at one instant; the viewer's brain interpolates between snapshots to perceive smooth motion. The CPU's job is to *place* dots as fast as possible, reading pre-computed addresses from the stack.

The 4-phase colour animation uses 50 frames per second for *colour* flexibility. Each frame displays one of four colour states; the viewer's retina averages them. No single frame contains the perceived result – it exists only in persistence of vision.

Both exploit the same physical reality: the CRT refreshes at 50Hz, and the human visual system cannot resolve individual frames at that rate. The Spectrum's *temporal* resolution is far richer than its spatial or colour resolution. Demoscene coders discovered that temporal resolution is the cheapest axis to exploit.

Both reduce their inner loops to the absolute minimum. The scroller to 36 T-states per dot. The colour animation to a single buffer copy per frame. Both move complexity out of the inner loop into pre-computation. And both produce results that look, to the casual viewer, like the hardware should not be capable of them.

This is what makes the demoscene a temporal art form. A screenshot of a dotfield scroller shows a scatter of pixels. A screenshot of a 4-phase colour animation shows two colours per cell, exactly as the hardware specifies. You have to see them *move* to see them work. The beauty is in the sequence, not the frame.

Practical 1: A Bouncing Dot-Matrix Text Scroller

Build a simplified dotfield scroller: a short text message rendered as a bouncing dot-matrix field using POP-based addressing.

Data structures. A page-aligned 8x8 bitmap font (the ROM font at `$3D00` works). A 256-byte sine table for the bounce offset. A RAM buffer for the address table (up to 4,096 x 2 bytes).

Table construction. Before each frame, iterate through the visible characters. For each bit in each font byte, calculate the screen address incorporating the sine-wave

bounce offset, and store it in the address table. This runs once per frame outside the inner loop.

Rendering. Disable interrupts. Save SP via self-modifying code. Point SP at the address table. Execute the unrolled inner loop: `ld a,(bc) : inc bc`, then 8 repetitions of `pop hl : rla : jr nc,skip : set N,(hl)` with N from 7 down to 0. Restore SP. Enable interrupts.

Main loop. `halt` (sync to 50Hz), clear the screen (PUSH-based clear from Chapter 3), build the address table, render the dotfield, advance scroll position and bounce phase.

Extensions. Partial screen clearing (track the bounding box). Double buffering via shadow screen on 128K. Multiple bounce harmonics. Variable dot density for a sparser, more ethereal look.

Practical 2: A 4-Phase Colour Cycling Animation

Build a 4-phase colour animation producing smooth gradients.

Pixel patterns. Fill bitmap memory with two complementary dither patterns. The simplest: even pixel lines get `$55` (01010101), odd lines get `$AA` (10101010). For production quality, use an ordered 4x4 Bayer matrix.

Attribute buffers. Pre-calculate four 768-byte buffers. Buffers 0 and 1 hold normal attributes with two different colour schemes (varying ink/paper across the screen for a diagonal gradient). Buffers 2 and 3 are the inverted versions – ink and paper bits swapped. The swap is a bit rotation: three RRCAs to move ink bits to paper position, three RLCAs the other way, mask and combine.

Main loop. Each frame: `halt`, index into a 4-entry table of buffer pointers using a phase counter (AND 3), LDIR 768 bytes to `$5800`, increment the phase counter. That is the entire runtime engine – about 16,000 T-states per frame.

Animation. For a moving gradient, regenerate one buffer per frame (the one about to become the oldest in the 4-frame cycle) with an advancing colour offset. This maintains a pipeline: display frame N while generating frame N+4. Alternatively, pre-compute all buffers across 128K banks for zero runtime cost.

Summary

- The **dotfield scroller** renders text as individual dots. The inner loop – `pop hl : rla : jr nc,skip : set ?,(hl)` – costs 36 T-states per opaque pixel, 26 per transparent pixel.
- **Stack-based addressing** encodes the bounce trajectory as pre-built screen addresses. POP retrieves them at 10 T-states each – the fastest random-access read on the Z80.
- **4-phase colour** cycles 4 attribute frames (2 normal + 2 inverted) at 50Hz. Persistence of vision averages the colours, creating the illusion of more than 2 colours per cell.

- The **inversion step** ensures all four colours contribute to every pixel position.
- Both techniques exploit **temporal resolution** to create effects impossible in any single frame.
- The scroller uses the stack for spatial flexibility; the colour animation uses frame alternation for colour flexibility – the two main axes of demoscene cheating.

Try It Yourself

1. Build the dotfield scroller. Start with a single static character plotted via the POP-based inner loop. Verify the expected timing with the border harness from Chapter 1. Then add the bounce table and watch it undulate.
2. Experiment with bounce parameters. Change the sine amplitude, spatial frequency, and phase speed. Small changes produce dramatic visual differences.
3. Build the 4-phase colour animation. Start with uniform colour (all cells the same in each phase). Verify you see a steady colour that is neither the ink nor the paper of any single frame. Then add the diagonal gradient.
4. Try different dithering patterns. Checkerboard, 2x2 blocks, Bayer matrix, random noise. Which minimise visible flicker? Which produce the smoothest perceived gradients?
5. Combine both techniques: 4-phase colour background with a monochrome dotfield scroller on top.

Sources: Introspec, “Technical Analysis of Illusion by X-Trade” (Hype, 2017); Introspec, “Making of Eager” (Hype, 2015); Dark, “Programming Algorithms” (Spectrum Expert #01, 1997). The inner loop disassembly and cycle counts follow Introspec’s 2017 analysis. The 4-phase colour technique is described in the Eager making-of article and the party version’s file_id.diz.

Chapter 11: Sound Architecture – AY, TurboSound, and Triple AY

"The AY chip has three voices. That's not a limitation – it's a design constraint that shaped an entire musical genre."

The AY-3-8910 is the voice of the ZX Spectrum 128K. General Instrument designed it in 1978 as a Programmable Sound Generator (PSG) – a single chip that could produce music and sound effects without a dedicated sound CPU. By the time Amstrad put it into the Spectrum 128K in 1986, it was already a proven workhorse: Intellivision, MSX, Atari ST, and dozens of arcade machines all used it or its pin-compatible variants (YM2149 in the Atari ST, AY-3-8912 with fewer I/O ports).

Fourteen registers. Three square-wave tone channels. One noise generator. One envelope generator. That is all you get. Everything you have ever heard in a Spectrum 128K chiptune – the pumping basslines, the rapid-fire arpeggiated chords, the snappy drums – comes from programming these fourteen registers fifty times per second.

This chapter takes you from bare register writes to a working music engine. By the end, you will understand exactly how a tracker player puts sound into the AY, and you will have the knowledge to write your own.

11.1 The AY-3-8910 Register Map

The AY has 14 registers, addressed R0 through R13. On the ZX Spectrum 128K, you access them through two I/O ports:

- **$FFFD** – Register select (write the register number here first)
- **$BFFD** – Data write (then write the value here)

Always select the register first, then write the data. This two-step process is fundamental:

```
z80 id:ch11_the_ay_3_8910_register_map ; Write value E to AY register A ay_write:
ld   bc, $FFFD     out  (c), a       ; select register     ld   b, $BF       ;
$BFFD high byte (C stays $FD)     out  (c), e       ; write data     ret
```

Note the trick: we only change the high byte of BC between the two OUT instructions. The low byte $FD stays in C throughout. This saves loading a full 16-bit value twice.

Complete Register Table

Reg	Name	Bits	Description
R0	Tone A period, low	8	Lower 8 bits of channel A tone period
R1	Tone A period, high	4	Upper 4 bits of channel A tone period
R2	Tone B period, low	8	Lower 8 bits of channel B tone period
R3	Tone B period, high	4	Upper 4 bits of channel B tone period
R4	Tone C period, low	8	Lower 8 bits of channel C tone period
R5	Tone C period, high	4	Upper 4 bits of channel C tone period
R6	Noise period	5	Noise generator period (0-31)
R7	Mixer / I/O enable	8	Tone and noise enable per channel + I/O
R8	Volume A	5	Channel A volume (0-15) or envelope mode
R9	Volume B	5	Channel B volume (0-15) or envelope mode
R10	Volume C	5	Channel C volume (0-15) or envelope mode
R11	Envelope period, low	8	Lower 8 bits of envelope period
R12	Envelope period, high	8	Upper 8 bits of envelope period
R13	Envelope shape	4	Envelope waveform shape (0-15)

Tone Channels (R0-R5): How Pitch Works

Each of the three tone channels produces a square wave. The frequency is controlled by a 12-bit period value split across two registers:

```
Frequency = AY_clock / (16 x period)
```

On the Spectrum 128K, the AY clock is 1.7734 MHz (half the CPU clock of 3.5469 MHz). So for middle C (approximately 262 Hz):

```
period = 1,773,400 / (16 x 262) = 423
```

The 12-bit period gives a range from 1 (110,837 Hz – inaudible ultrasonic) to 4095 (27 Hz – a deep bass rumble). Here is a practical note table for octave 4:

Note	Frequency (Hz)	Period (decimal)	R_LO	R_HI
C4	261.6	424	$A8	$01
C#4	277.2	400	$90	$01
D4	293.7	378	$7A	$01
D#4	311.1	357	$65	$01

Note	Frequency (Hz)	Period (decimal)	R_LO	R_HI
E4	329.6	337	$51	$01
F4	349.2	318	$3E	$01
F#4	370.0	300	$2C	$01
G4	392.0	283	$1B	$01
G#4	415.3	267	$0B	$01
A4	440.0	252	$FC	$00
A#4	466.2	238	$EE	$00
B4	493.9	225	$E1	$00
C5	523.3	212	$D4	$00

To go up one octave, halve the period. To go down, double it. The full note table for all useful octaves lives in Appendix G.

The Noise Generator (R6)

Register R6 controls a single noise generator shared by all three channels. The 5-bit value (0-31) sets the noise "pitch" – lower values produce higher, hissier noise; higher values produce lower, rougher noise. The noise generator produces pseudo-random output using a 17-bit linear feedback shift register.

Useful ranges: - **0-5**: High hiss (hi-hat, cymbal) - **6-12**: Mid noise (snare body) - **13-20**: Low rumble (explosion) - **21-31**: Very low (wind, thunder)

R7: The Mixer – The Most Important Register

Register R7 is the heart of the AY. Six bits control which channels receive tone, noise, or both. Two more bits control the I/O port direction (irrelevant for sound, leave them as inputs = 1).

```
text id:ch11_r7_the_mixer_the_most Bit 7: I/O port B direction (1 = input) Bit
6: I/O port A direction (1 = input) Bit 5: Noise C enable (0 = ON, 1 = off) Bit
4: Noise B enable (0 = ON, 1 = off) Bit 3: Noise A enable (0 = ON, 1 = off) Bit
2: Tone C enable  (0 = ON, 1 = off) Bit 1: Tone B enable  (0 = ON, 1 = off) Bit
0: Tone A enable  (0 = ON, 1 = off)
```

The confusing part: 0 means ON. This trips up everyone the first time. The AY uses active-low logic for the mixer enables. A cleared bit enables the corresponding source.

Here is a table of useful mixer combinations:

Value	Binary (NcNbNa TcTbTa)	Effect
$38	`111 000`	All three tones on, no noise
$3E	`111 110`	Tone A only
$3D	`111 101`	Tone B only
$3B	`111 011`	Tone C only
$36	`110 110`	Tone A + Noise A
$07	`000 111`	All three noise, no tones
$30	`110 000`	Tones A+B+C, Noise A
$00	`000 000`	Everything on (cacophony)

Value	Binary (NcNbNa TcTbTa)	Effect
$3F	111 111	Everything off (silence)

A common pattern for music: tones A+B for melody and harmony, tone C + noise C for drums:

```
z80 id:ch11_r7_the_mixer_the_most_2 ; Mixer: tone A on, tone B on, tone C on,
noise C on ; Binary: 00 011 000 = noise C on (bit5=0) + all tones on (bits0-2=0)
;         noise A,B off (bits3,4=1), I/O bits=0 ; = $18 ld   a, R_MIXER ld   e,
$18 call ay_write
```

Volume Registers (R8-R10)

Each channel has a 4-bit volume control (0-15, where 15 is loudest). But bit 4 is special: setting it switches that channel to **envelope mode**, where the volume is controlled automatically by the envelope generator instead of the fixed value.

```
text id:ch11_volume_registers_r8_r10 Bit 4: 1 = use envelope generator, 0 = use
bits 3-0 Bits 3-0: Fixed volume level (0-15)
```

The volume curve is logarithmic on a real AY-3-8910 (each step is approximately 1.5 dB) but varies between chip revisions and clones. The YM2149 has a different volume curve, which is why the same tune sounds subtly different on an Atari ST versus a Spectrum.

Envelope Generator (R11-R13)

The AY has one envelope generator – shared across all channels that use it. It automatically modulates the volume according to a repeating waveform.

R11-R12: Envelope Period – A 16-bit value controlling the envelope speed:

```
Envelope_frequency = AY_clock / (256 x period)
```

At period = 1, the envelope cycles at about 6,927 Hz. At period = 65535, it cycles at roughly 0.11 Hz – about once every 9 seconds.

R13: Envelope Shape – Four bits select the waveform. Although 4 bits allow 16 values, only 10 shapes are unique:

Value	Shape	Description
$00-$03	___	Single decay, then silent. (All four are identical.)
$04-$07	/___	Single attack, then silent. (All four are identical.)
$08	\\\\	Repeating sawtooth down.
$09	___	Single decay, then silent. (Same as $00.)
$0A	\/\/	Repeating triangle (decay-attack).
$0B	\'''	Single decay, then hold at max.
$0C	////	Repeating sawtooth up.
$0D	/'''	Single attack, then hold at max.

Value	Shape	Description
$0E $0F	/\/\ /___	Repeating triangle (attack-decay). Single attack, then silent. (Same as $04.)

Waveform diagrams (volume over time):

```
$00-$03, $09:  |\        $04-$07, $0F:    /|
               | \                       / |
               |  \___                  /  |___

$08: |\  |\  |\           $0C:   /|  /|  /|
     | \ | \ | \                 / | / | / |
     |  \| \| \|                /  |/  |/  |

$0A: |\  /\  /\            $0E:   /\  /\  /\
     | \/  \/  \                  /  \/  \/  \/
     |          |               |

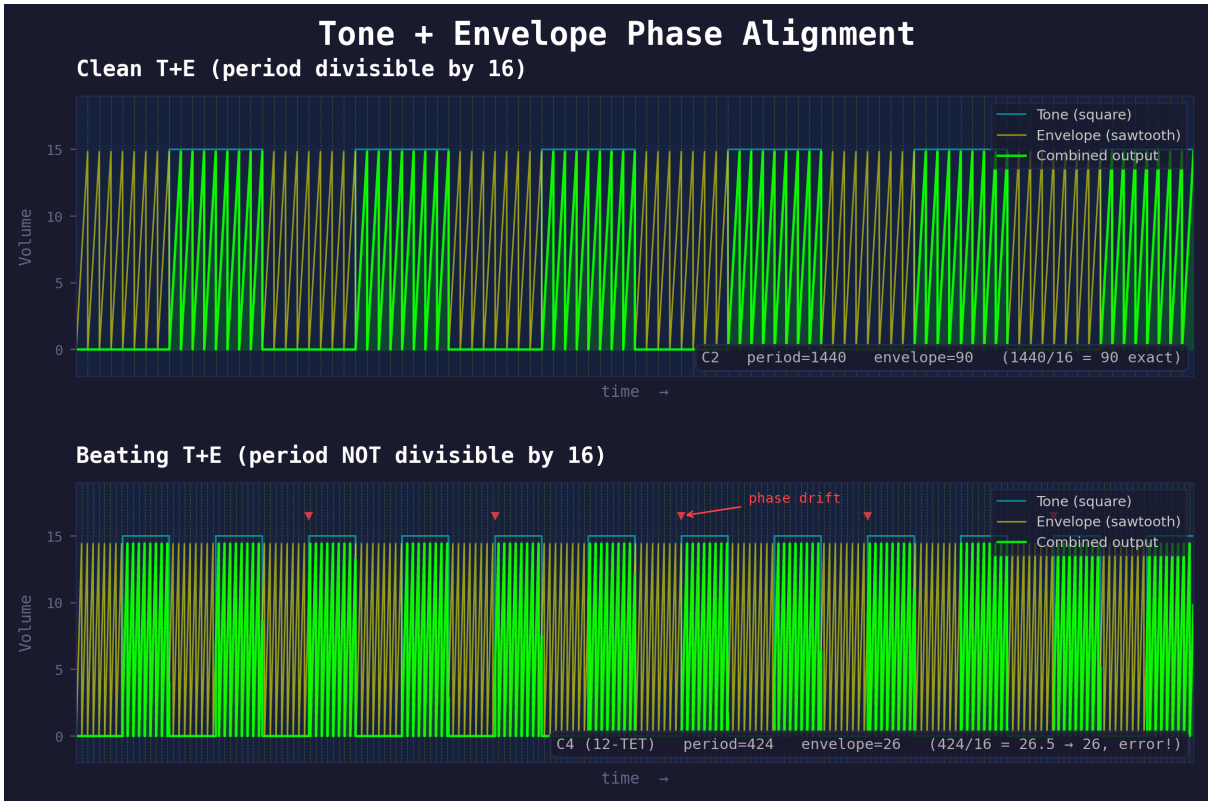
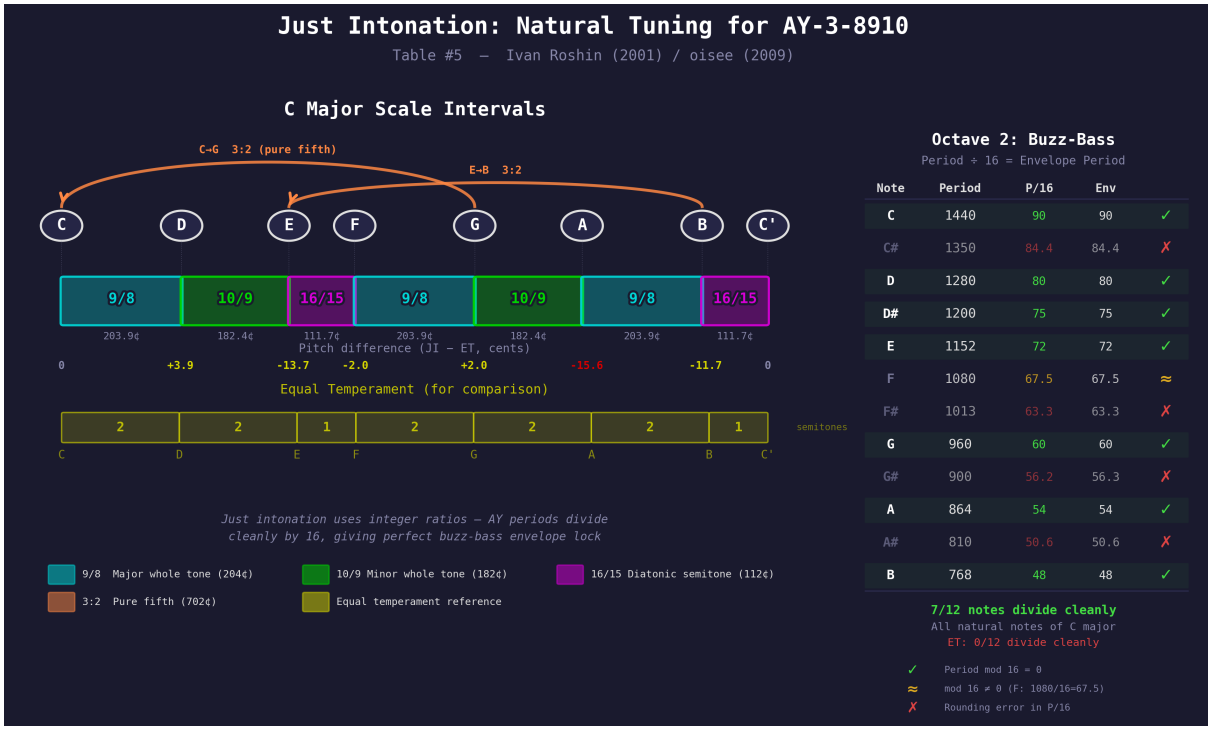
$0B: |\                    $0D:    /|
     | \                         / |
     |  ''''''''               /  |''''''''
```

Key insight: Writing to R13 *restarts* the envelope from the beginning. This is crucial for bass techniques – you can trigger a new envelope cycle at any time by writing to R13, even with the same value.

11.2 Chiptune Techniques on 3 Channels

Three channels is not a lot. A real band has bass, drums, melody, and harmony as a minimum. Chiptune composers developed ingenious tricks to create the illusion of more:

Arpeggios: Fake Chords

An arpeggio rapidly cycles through the notes of a chord – one note per frame. At 50 Hz (PAL), cycling through C4, E4, and G4 every frame produces an effect that the ear perceives as a C major chord, even though only one note sounds at any instant.

“‘z80 id:ch11_arpeggios_fake_chords ; Arpeggio: cycle C-E-G on channel A every frame ; Called once per frame from the interrupt handler arpeggio: ld a, (arp_pos) inc a cp 3 jr nz, .no_wrap xor a .no_wrap: ld (arp_pos), a

```
; Index into note table
ld   hl, arp_notes
add  a, a            ; x2 (each entry is a 16-bit period)
ld   e, a
ld   d, 0
add  hl, de
ld   e, (hl)
```



Figure 6: AY-3-8910 register map



Figure 7: AY envelope shape waveforms

```
inc  hl
ld   d, (hl)

; Write period to channel A
ld   a, R_TONE_A_LO
call ay_write_de_lo
ld   a, R_TONE_A_HI
call ay_write_de_hi
ret
```

arp_pos: DB 0 arp_notes: DW 424, 337, 283 ; C4, E4, G4

```
In a tracker, arpeggios are notated as effects applied to notes. Vortex Tracker II uses orname
tick semitone offsets.

### Buzz-Bass: The Envelope Trick

Here is the most distinctive chiptune bass sound: the "buzz." It works by abusing the envelope

```z80 id:ch11_buzz_bass_the_envelope_trick
; Buzz-bass: play bass note using envelope generator
; DE = envelope period for desired note
buzz_bass:
    ; Set envelope period
    ld   a, R_ENV_LO
    call ay_write_de_lo
    ld   a, R_ENV_HI
    call ay_write_de_hi

    ; Set envelope shape to repeating sawtooth down
```
```

```
    ; Writing R13 restarts the envelope
    ld   a, R_ENV_SHAPE
    ld   e, $08          ; \\\\  repeating sawtooth down
    call ay_write

    ; Set channel volume to envelope mode (bit 4 = 1)
    ld   a, R_VOL_C
    ld   e, $10          ; bit 4 set = use envelope
    call ay_write
    ret
```

The envelope period for a given bass note is:

```
envelope_period = AY_clock / (256 x desired_frequency)
```

For a bass C2 (65.4 Hz): period = 1,773,400 / (256 x 65.4) = 106.

The buzz-bass gives you a bass instrument that sounds fundamentally different from the tone channels, effectively adding a fourth voice to your arrangement.

The Period Alignment Problem

There is a catch with buzz-bass that equal-tempered note tables hide from you. Look at the formula again:

```
tone_period    = AY_clock / (16 x frequency)
envelope_period = tone_period / 16
```

For the buzz to sound clean, the envelope period must be *exactly* `tone_period / 16`. But integer division truncates. If the tone period is not divisible by 16, the envelope period has a rounding error – and the envelope waveform drifts against the tone, producing audible beating.

Check our standard table. Octave 4:

Note	Period	Period mod 16	Envelope = Period / 16	Error?
C4	424	8	26 (should be 26.5)	Yes
D4	378	10	23 (should be 23.625)	Yes
E4	337	1	21 (should be 21.0625)	Yes
F4	318	14	19 (should be 19.875)	Yes
G4	283	11	17 (should be 17.6875)	Yes
A4	252	12	15 (should be 15.75)	Yes
B4	225	1	14 (should be 14.0625)	Yes

Not a single clean division! Every note in the equal-tempered scale produces a slightly detuned envelope. For short percussive buzz sounds the beating is masked, but for sustained bass notes it creates an unpleasant warble.

Natural Tuning: Table #5

In June 2001, Ivan Roshin published “Частотная таблица с нулевой погрешностью” (A Frequency Table with Zero Error), arriving at the same conclusion that centuries of music theory had already established: replace equal temperament with *just intonation* – integer-ratio intervals that the AY hardware can divide cleanly.



Figure 8: Tone + Envelope Phase Alignment: clean T+E with period divisible by 16 (top) vs beating T+E with rounding error (bottom)

The natural scale for C major / A minor uses these intervals:

```
C [9/8] D [10/9] E [16/15] F [9/8] G [10/9] A [9/8] B [16/15] C
```

This gives pure fifths (3:2 ratio) for C–G, E–B, A–E. Chromatic notes (sharps/flats) are calculated with the 16/15 ratio.

The resulting periods, computed for a *non-standard* AY clock of 1,520,640 Hz:

```
z80 id:ch11_natural_tuning_table_5_2 ; Table #5: Natural tuning for AY clock =
1,520,640 Hz ; 96 notes (8 octaves), C major / A minor ; Ivan Roshin (concept,
2001), oisee/siril (VTi implementation, 2009) natural_note_table:     ; Octave
1: every period divisible by 16!     DW 2880, 2700, 2560, 2400, 2304, 2160
DW 2025, 1920, 1800, 1728, 1620, 1536     ; Octave 2     DW 1440, 1350, 1280,
1200, 1152, 1080     DW 1013,  960,  900,  864,  810,  768     ; Octave 3     DW
720,  675,  640,  600,  576,  540     DW  506,  480,  450,  432,  405,  384
; Octave 4     DW  360,  338,  320,  300,  288,  270     DW  253,  240,  225,
216,  203,  192     ; Octave 5     DW  180,  169,  160,  150,  144,  135     DW
127,  120,  113,  108,  101,   96     ; Octave 6     DW   90,   84,   80,   75,
72,   68     DW   63,   60,   56,   54,   51,   48     ; Octave 7     DW   45,
42,   40,   38,   36,   34     DW   32,   30,   28,   27,   25,   24     ;
Octave 8     DW   23,   21,   20,   19,   18,   17     DW   16,   15,   14,
14,   13,   12
```

The key insight is that most main-scale periods are now divisible by 16. Here is octave 2 – the bass range that matters most for buzz:



Figure 9: Just Intonation: interval structure of the natural scale with period divisibility table for buzz-bass

Note	Period	mod 16	Envelope = Period/16	Clean?
C2	1440	0	90	Yes
C#2	1350	6	84.375 → 84	No
D2	1280	0	80	Yes
D#2	1200	0	75	Yes
E2	1152	0	72	Yes
F2	1080	8	67.5 → 68	~
F#2	1013	5	63.3 → 63	No
G2	960	0	60	Yes
G#2	900	4	56.25 → 56	No
A2	864	0	54	Yes
A#2	810	10	50.6 → 51	No
B2	768	0	48	Yes

Seven of twelve notes divide cleanly – all the natural notes of C major. Compare to the equal-tempered table where *none* do. On those seven notes the envelope and tone generators lock in phase, and the buzz-bass sounds pure.

The tradeoff: this table is only correct for C major / A minor. To play in other keys, you change the AY clock frequency:

Key	Chip Frequency (Hz)
C/Am	1,520,640
C#/A#m	1,611,062
D/Bm	1,706,861

Key	Chip Frequency (Hz)
D#/Cm	1,808,356
E/C#m	1,915,886
F/Dm	2,029,811
F#/D#m	2,150,510
G/Em	2,278,386
G#/Fm	2,413,866
A/F#m	2,557,401
A#/Gm	2,709,472
B/G#m	2,870,586

On real hardware the AY clock is fixed, so you cannot actually change keys at runtime. But in an emulator or tracker like Vortex Tracker II, the “chip frequency” is a setting. This is exactly what the Vortex Tracker Improved (VTi) modification by oisee did in 2009: it added Table #5 (the fifth table, index 4 counting from zero) with these natural periods, plus a per-module chip frequency setting that selects the key.

The autosiril MIDI-to-PT3 converter defaults to Table #5 precisely because of these clean envelope ratios – most converted tracks use buzz-bass extensively, and the natural tuning eliminates beating.

In practice: if you are writing a tracker module that relies heavily on buzz-bass, consider composing in C/Am with Table #5. The envelopes will lock perfectly to the tone. If you need a different key, either transpose the chip frequency (tracker-side) or accept the small rounding errors of equal temperament. For short percussive buzz sounds, the difference is inaudible; for sustained bass drones, it is very noticeable.

Drum Synthesis

With only one noise generator, drums need to share. The standard approach:

Snare drum: Noise channel + rapid envelope decay.

“‘z80 id:ch11_drum_synthesis ; Snare: noise burst with fast decay drum_snare: ; Noise period: mid-range ld a, R_NOISE ld e, 8 call ay_write

```
; Mixer: enable noise on channel C
ld   a, R_MIXER
ld   e, $18          ; tones A+B+C on, noise C on
call ay_write

; Short envelope: fast decay
ld   a, R_ENV_LO
ld   e, 200
call ay_write
ld   a, R_ENV_HI
ld   e, 0
call ay_write

; Envelope shape: single decay
```

```
ld   a, R_ENV_SHAPE
ld   e, $00          ; \___  single decay to silence
call ay_write

; Channel C to envelope mode
ld   a, R_VOL_C
ld   e, $10
call ay_write
ret


**Kick drum:** Rapid tone sweep downward. Set channel C to a low tone period, then increase th

```z80 id:ch11_drum_synthesis_2
; Kick: tone sweep down over 4 frames
; Call once per frame while kick_counter > 0
kick_update:
    ld   a, (kick_counter)
    or   a
    ret  z

    dec  a
    ld   (kick_counter), a

    ; Sweep tone down (increase period)
    ld   hl, (kick_period)
    ld   de, 40          ; sweep speed
    add  hl, de
    ld   (kick_period), hl

    ; Write to channel C
    ld   a, R_TONE_C_LO
    ld   e, l
    call ay_write
    ld   a, R_TONE_C_HI
    ld   e, h
    call ay_write
    ret

kick_counter: DB 0
kick_period:  DW 0
```

Hi-hat: Very short noise burst, high noise frequency (low R6 value), immediate volume cut after 1-2 frames.

Ornaments: Per-Frame Modulation

An ornament is a table of per-frame offsets applied to the pitch or volume of a note. Ornaments give life to otherwise static square waves: vibrato, pitch slides, tremolo, attack/decay envelopes – all accomplished by table lookup.

```
z80 id:ch11_ornaments_per_frame ; Example ornament: pitch vibrato ; Table of
signed semitone offsets, applied once per frame ornament_vibrato:     DB 0, 0,
```
```

```
1, 1, 0, 0, -1, -1    ; 8-frame cycle     DB $80                              ; end
marker
```

The player engine applies the ornament offset to the base note's period value each frame, producing smooth modulation.

11.3 TurboSound: 2 x AY

Pentagon and Scorpion clones introduced TurboSound – two AY chips in one machine. The second chip is addressed by selecting it through a bit pattern written to port $FFFD before register operations.

Chip Selection

On the NedoPC TurboSound card (the most common modern design), chip selection works through port $FFFD:

"'z80 id:ch11_chip_selection ; Select chip 0 (primary) ld bc, $FFFD ld a, $FF ; bit pattern: select chip 0 out (c), a

; Select chip 1 (secondary) ld bc, $FFFD ld a, $FE ; bit pattern: select chip 1 out (c), a

```
Once a chip is selected, all subsequent register reads and writes via $FFFD/$BFFD go to that c

### 6 Channels, True Stereo

TurboSound doubles everything: 6 tone channels, 2 independent noise generators, 2 independent

| Chip | Channels | Stereo | Role |
|------|----------|--------|------|
| Chip 0 | A0, B0, C0 | Left or center | Lead melody, harmony, bass |
| Chip 1 | A1, B1, C1 | Right or center | Counter-melody, pads, drums |

Or mix them for a wide stereo field:
- Bass on both chips (center)
- Lead on chip 0 (left)
- Counter-melody on chip 1 (right)
- Drums split: kick on chip 0, snare/hi-hat on chip 1

What TurboSound changes musically is significant: on a single AY, the composer is constantly m
stealing. With TurboSound, you have room. Dedicated bass channel, dedicated drums, and four re

### Engine Modification

Adapting a single-AY engine to TurboSound is straightforward. Your interrupt handler becomes:

```z80 id:ch11_engine_modification
music_frame:
    ; Update chip 0
```
```

```
    ld   a, $FF
    ld   bc, $FFFD
    out  (c), a          ; select chip 0
    call update_chip0    ; write 14 registers

    ; Update chip 1
    ld   a, $FE
    ld   bc, $FFFD
    out  (c), a          ; select chip 1
    call update_chip1    ; write 14 registers
    ret
```

The register write loop writes all 14 registers from a buffer in RAM. For two chips, you maintain two 14-byte buffers and blast them out sequentially. Each register write requires selecting the register (LD A,reg + OUT) then writing the value (LD A,val + OUT), costing about 36 T-states per register. For 28 registers total: approximately 1,008 T-states, plus chip selection overhead.

11.4 Triple AY on ZX Spectrum Next

The ZX Spectrum Next takes it further: three AY-compatible sound chips, giving you 9 channels. But the Next's implementation goes beyond simple triplication.

Enhanced Features

The Next's AY chips include **per-channel stereo panning**. Each channel can be individually panned left, right, or center – something the original AY never supported. This is controlled through additional Next-specific registers.

The three chips are addressed via Next register $06 (peripheral 2 setting) or through the standard $FFFD port with chip select values.

9 Channels: Orchestral Thinking

Nine channels fundamentally change how you approach composition on 8-bit hardware. Instead of clever tricks to simulate complexity, you can think orchestrally:

Channel	Assignment	Panning
Chip 0, A	Bass line	Center
Chip 0, B	Rhythm guitar / pad	Left
Chip 0, C	Lead melody	Center
Chip 1, A	Harmony / counter-melody	Right
Chip 1, B	Arpeggio chord	Left
Chip 1, C	Percussion: kick + snare	Center
Chip 2, A	Hi-hat / cymbal	Right
Chip 2, B	Sound effects (reserved)	Center
Chip 2, C	Ambient / pad layers	Left

This is enough for genuinely rich arrangements. You have a dedicated SFX channel that never interrupts the music. You have independent noise generators for layered percussion. You can sustain chords without arpeggios. The AY's character remains – square waves are still square waves – but the compositional freedom approaches that of an Amiga MOD tracker with its four sample channels.

11.5 Music Engine Architecture

A music engine is the code that reads pattern data and writes AY registers at the correct tempo. On the Spectrum, it lives inside the interrupt handler.

The Interrupt-Driven Player

The ZX Spectrum's IM2 (Interrupt Mode 2) fires once per frame – every 1/50th of a second on PAL systems. The music engine hooks into this:

"'z80 id:ch11_the_interrupt_driven_player ; Setup: install IM2 handler setup_im2: di ld a, $C0 ; interrupt vector table at $C000 ld i, a im 2

```
; Fill vector table at $C000 with $C1C1
; Handler at $C1C1
ld   hl, $C000
ld   de, $C001
ld   bc, 256
ld   (hl), $C1
ldir

; Place JP at $C1C1
ld   a, $C3          ; JP opcode
ld   ($C1C1), a
ld   hl, isr_handler
ld   ($C1C2), hl

ei
ret
```

isr_handler: push af push bc push de push hl ; ... push all registers you use ...

```
call music_play      ; <-- the player routine

; ... pop all registers ...
pop  hl
pop  de
pop  bc
pop  af
ei
reti


### The Player Loop

The player routine, called 50 times per second, does the following:
```

```
1. Decrement the current note's duration counter
2. If zero, advance to the next note in the pattern
3. Apply ornaments and effects (arpeggio, vibrato, slide) to each channel
4. Calculate final period and volume for each channel
5. Write all 14 AY registers

A simplified skeleton:

```z80 id:ch11_the_player_loop
music_play:
    ; Decrement speed counter
    ld   a, (speed_counter)
    dec  a
    ld   (speed_counter), a
    ret  nz              ; not time for a new row yet

    ; Reset speed counter
    ld   a, (song_speed)
    ld   (speed_counter), a

    ; Process each channel
    ld   ix, channel_a_data
    call process_channel
    ld   ix, channel_b_data
    call process_channel
    ld   ix, channel_c_data
    call process_channel

    ; Write all registers to AY
    call ay_flush_registers
    ret
```

Frame Budget

Here is the critical question: how many T-states can the music engine consume before it starves the main program?

The frame is 71,680 T-states on Pentagon (69,888 on 48K). The interrupt fires at the start of the frame. If the music player takes 5,000 T-states, the main program has 66,680 left for visuals.

Typical costs: - **Simple player** (no effects, no ornaments): ~1,500-2,500 T-states - **Pro Tracker 3 player**: ~3,000-5,000 T-states - **Full Vortex Tracker II player** with ornaments and effects: ~4,000-7,000 T-states - **TurboSound player** (2 chips): ~6,000-10,000 T-states

For a demo running a cycle-hungry effect, 7,000 T-states for music is significant – about 10% of the frame. Plan accordingly.
```

Formats and Trackers

The modern standard for AY music on the Spectrum is **Vortex Tracker II** (.pt3 format). It is a cross-platform tracker that runs on Windows and outputs files directly playable by proven Z80 player routines.

Format	Tracker	Features	Player Size
.pt3	Vortex Tracker II / Pro Tracker 3	Ornaments, samples, effects	~1.2-1.8 KB
.asc	ASC Sound Master	Simpler, smaller player	~0.8-1.0 KB
.sqt	SQ-Tracker	Compact, good compression	~0.6-0.8 KB
.stc	Sound Tracker	Basic, oldest	~0.5-0.7 KB

The Pipeline:

1. Compose in Vortex Tracker II on your PC
2. Export as .pt3 file
3. Include the .pt3 player source (Z80 assembly) in your project
4. Include the .pt3 data file as a binary blob
5. Call `music_init` at startup (passing the address of the song data)
6. Call `music_play` from your interrupt handler every frame

“‘z80 id:ch11_formats_and_trackers ; In your main code: ld hl, song_data call music_init

; In your interrupt handler: call music_play

; At the end of the binary: song_data: INCBIN “mysong.pt3”

```
This is the standard approach used by virtually every Spectrum 128K demo and game since the ea

---

## 11.6 Sound Effects System

Games need sound effects, and sound effects need channels. The standard approach is **priority
based channel stealing**: when a sound effect triggers, it temporarily takes over a channel fr

### Channel Stealing

```z80 id:ch11_channel_stealing
; Trigger a sound effect on the SFX channel
; HL = pointer to SFX data table
sfx_trigger:
    ld   (sfx_pointer), hl
    ld   a, 1
    ld   (sfx_active), a
    ld   a, 0
    ld   (sfx_frame), a
    ret
```
```

```
; Called every frame from the interrupt handler, AFTER music_play
sfx_update:
    ld   a, (sfx_active)
    or   a
    ret  z               ; no active SFX

    ; Read current SFX frame data
    ld   hl, (sfx_pointer)
    ld   a, (sfx_frame)
    ld   e, a
    ld   d, 0

    ; Each SFX frame: [tone_lo, tone_hi, noise, volume, mixer_mask]
    ; 5 bytes per frame
    push hl
    ld   b, 5
    call multiply_de_b   ; DE = frame * 5 (or use repeated add)
    pop  hl
    add  hl, de

    ld   a, (hl)
    cp   $FF             ; end marker?
    jr   z, .sfx_done

    ; Override channel C with SFX data
    ld   e, (hl) : inc hl
    ld   a, R_TONE_C_LO
    call ay_write

    ld   e, (hl) : inc hl
    ld   a, R_TONE_C_HI
    call ay_write

    ld   e, (hl) : inc hl
    ld   a, R_NOISE
    call ay_write

    ld   e, (hl) : inc hl
    ld   a, R_VOL_C
    call ay_write

    ; Advance frame counter
    ld   a, (sfx_frame)
    inc  a
    ld   (sfx_frame), a
    ret

.sfx_done:
    xor  a
    ld   (sfx_active), a ; deactivate SFX
```

```
    ret
```

Procedural SFX Tables

Sound effects are defined as tables of per-frame register values. Here are four classic game sounds:

Explosion: Noise with decaying volume.

```
z80 id:ch11_procedural_sfx_tables sfx_explosion:     ; tone_lo, tone_hi, noise_period,
volume, (unused)     DB 0, 0, 15, 15, 0    ; frame 0: loud low noise     DB 0,
0, 18, 13, 0    ; frame 1     DB 0, 0, 20, 11, 0    ; frame 2     DB 0, 0, 22,
9,  0    ; frame 3     DB 0, 0, 25, 7,  0    ; frame 4     DB 0, 0, 28, 5,  0
; frame 5     DB 0, 0, 30, 3,  0    ; frame 6     DB 0, 0, 31, 1,  0    ; frame
7     DB $FF                  ; end
```

Laser: Fast tone sweep downward.

```
z80 id:ch11_procedural_sfx_tables_2 sfx_laser:     DB 10, 0, 0, 14, 0    ; frame
0: high tone     DB 30, 0, 0, 13, 0    ; frame 1: sweeping down     DB 60, 0,
0, 12, 0    ; frame 2     DB 100,0, 0, 10, 0    ; frame 3     DB 160,0, 0, 8,  0
; frame 4     DB 240,0, 0, 5,  0    ; frame 5     DB 200,1, 0, 3,  0    ; frame
6: into low range     DB $FF                 ; end
```

Jump: Short tone sweep upward.

```
z80 id:ch11_procedural_sfx_tables_3 sfx_jump:     DB 200,0, 0, 12, 0    ; frame
0: mid tone     DB 150,0, 0, 11, 0    ; frame 1: rising     DB 100,0, 0, 10, 0
; frame 2     DB 60, 0, 0, 8,  0    ; frame 3: high     DB 40, 0, 0, 5,  0    ;
frame 4     DB $FF                 ; end
```

Pickup: Rapid arpeggio upward.

```
z80 id:ch11_procedural_sfx_tables_4 sfx_pickup:     DB $FC,0, 0, 14, 0    ;
frame 0: A4     DB $D4,0, 0, 13, 0    ; frame 1: C5     DB $A0,0, 0, 12, 0    ;
frame 2: E5 (approx)     DB $6A,0, 0, 11, 0    ; frame 3: C6 (approx)     DB
$6A,0, 0, 8,  0    ; frame 4: sustain     DB $6A,0, 0, 4,  0    ; frame 5: fade
DB $FF                 ; end
```

11.7 Putting It Together: The Working Example

The file `chapters/ch11-sound/examples/ay_test.a80` contains a complete, assembling example that demonstrates the fundamentals: initializing the AY, setting the mixer, writing tone periods, and playing a melody. Study it alongside this chapter – every concept discussed here is exercised in that code.

The key patterns to notice in the example:

1. **Two-step port access**: Select register via $FFFD, then write data via $BFFD.
2. **Mixer configuration**: $3E enables only tone A (binary `111 110` – remember, 0 = ON).
3. **Note table**: Pre-calculated period values for each pitch.
4. **HALT-based timing**: Each `HALT` waits for one interrupt, giving 50 Hz timing resolution.

To extend this example into a real music player, you would replace the linear melody table with pattern-based data, add ornament processing, and move the playback into an IM2 interrupt handler so the main loop is free for visuals.

Sidebar: Beeper – A Brief History of Impossibility

Before the 128K and its AY chip, the original 48K Spectrum had exactly one bit of audio output. Pin 4 of port $FE. High or low. That is it.

One bit means one square wave at whatever frequency you toggle it. No volume control, no mixing, no hardware help. To play a note, you sit in a tight loop toggling the bit at the right frequency. To play *two* notes, you interleave two toggle loops. To play three, you interleave three. Each additional voice eats CPU time that could be doing something else – like, say, drawing graphics.

And yet.

Between 2010 and 2015, Shiru (Shiru Otaku) cataloged approximately 30 distinct beeper engines, each using a different technique to extract polyphony from a single bit. The approaches ranged from simple pin-compatible pulse interleaving (2-3 channels) to extraordinary feats of engineering:

- **ZX-16** by Jan Deak: 16-channel polyphony on a single bit. Sixteen. The CPU does nothing but toggle the speaker bit using a carefully timed schedule that approximates the sum of 16 independent waveforms through pulse-density modulation.
- **Octode XL**: 8-channel beeper engine that actually leaves enough CPU for visuals.
- **Rain** by Life on Mars (2016): A full demo running a 9-channel beeper engine *simultaneously with visual effects* on a 48K Spectrum. The entire production – music and graphics – runs without an AY chip, without bank switching, without anything beyond the base machine.

These are among the most remarkable engineering achievements in 8-bit computing. They prove that limits are more permeable than they appear. But they are also impractical for general use: most beeper engines consume 50-90% of CPU time, leaving almost nothing for gameplay or effects. The AY chip exists precisely to offload sound generation to dedicated hardware.

We cover beeper engines here as historical context and inspiration. For practical music in your demos and games, the AY is where you should focus your effort.

Sidebar: Agon Light 2 – VDP Sound System

The Agon Light 2 takes a completely different approach to sound. Its audio is generated by the ESP32 co-processor (the VDP), not by a dedicated sound chip. You send VDU commands over the serial link, and the ESP32 synthesizes audio in software.

Waveforms: The Agon's sound system offers multiple waveform types per channel – square, sine, triangle, sawtooth, and noise. This is already more flexible than the AY's square-only tone generators.

ADSR Envelopes: Each channel has a fully programmable Attack-Decay-Sustain-Release envelope. No sharing – every channel gets its own independent envelope, unlike the AY's single shared envelope generator.

Channel Count: The VDP audio system supports multiple simultaneous channels (the exact number depends on the firmware version, but typically 8 or more).

The Trade-off: The Agon's sound is controlled through VDU byte sequences sent over serial. This means: - Higher latency than direct register writes (serial transfer time) - Less precise timing (you cannot bit-bang exact T-state-level sync) - But much less CPU overhead (the eZ80 just sends commands; the ESP32 does all synthesis)

The paradigm is closer to MIDI than to register-level programming. You tell the VDP "play this note on channel 3 with a sine wave and this ADSR envelope," and it handles the rest. The musical goals are the same – melody, bass, drums, effects – but the programming model is fundamentally different. No mixer register, no period calculations, no envelope shape tables. Just commands and parameters.

For cross-platform projects, consider abstracting your sound system behind a common API: `sound_play_note(channel, note, instrument)`. On the Spectrum, the instrument lookup writes AY registers. On the Agon, it sends VDU commands. Same music data, different backends.

11.8 Practical Exercises

Exercise 1: Register Explorer. Write a program that lets you modify any AY register in real time using keyboard input. Display all 14 register values on screen. This is your single most useful debugging tool for sound work.

Exercise 2: Three-Channel Arrangement. Compose a simple 16-bar tune using all three channels: melody on A, bass (buzz-bass using envelope) on C, and arpeggiated chords on B. Use the HALT-based timing from the example as a starting point, then refactor it into an IM2-driven player.

Exercise 3: Drum Kit. Implement four drum sounds (kick, snare, hi-hat, crash) as procedural SFX tables. Write a simple drum pattern that plays them in sequence. Integrate with the tune from Exercise 2.

Exercise 4: Vortex Tracker Integration. Download Vortex Tracker II, compose a short tune, export as .pt3, and integrate the standard .pt3 player into a Spectrum

program. Verify it plays correctly in an emulator.

Summary

The AY-3-8910 is simple on paper: 14 registers, 3 channels, basic waveforms. But the gap between "simple" and "limited" is filled by technique. Arpeggios fake chords. Envelope abuse creates bass. Noise shaping synthesizes drums. Ornaments breathe life into static tones. And when three channels are genuinely not enough, TurboSound doubles them and the Next triples them.

The architecture pattern is consistent across all configurations: an interrupt fires 50 times per second, a player routine reads pattern data and calculates register values, and those values are blasted to the AY in a tight loop. Whether you are writing your own player or integrating Vortex Tracker, the flow is the same. Understanding the registers means understanding the sound.

In the next chapter, we will put this to work: digital drum samples blended with AY playback, frame-accurate synchronization between music and visuals, and the scripting engines that tie a demo together.

> **Sources:** Dark "GS Sound System" (Spectrum Expert #01, 1997); Dark "Music" (Spectrum Expert #02, 1998); Shiru "Beeper 20XX" (Hype 2016); Rain file_id.diz; Info Guide #14 ASC Sound Master docs (ZXArt 2024); Vortex Tracker II documentation

Chapter 12: Digital Drums and Music Sync

> *"My brain is not coping with asynchronous coding well."* – Introspec, file_id.diz for the party version of Eager (to live), 3BM Open Air 2015

A demo is not a slideshow of effects. A demo is a performance – one where every visual event lands on the beat, every transition breathes with the music, and the audience never suspects that behind the curtain, a 3.5MHz processor is juggling half a dozen competing demands with no operating system, no threads, and no safety net.

This chapter is about the architecture that makes that juggling act possible. We have spent the previous chapters building individual effects – tunnels, zoomers, scrollers, colour animations – and in Chapter 11 we learned how the AY chip produces music. Now we must wire everything together. The questions are no longer "how do I draw a tunnel?" or "how do I play a note?" but rather: How do I play a drum sample that consumes nearly all the CPU while keeping the visuals smooth? How do I synchronise effect changes to the beat of the music? How do I structure a two-minute demo so that it runs reliably from start to finish?

The answers come from three sources. Introspec's Eager (2015) gives us digital drum synthesis and asynchronous frame generation. diver4d's GABBA (2019) shows a radically different approach to music sync using a video editor as a timeline tool. And Robus's threading system (2015) demonstrates that honest multithreading on the Z80 is possible, if rarely necessary.

Together, these three techniques represent the architectural thinking that separates a collection of effects from a finished demo.

12.1 Digital Drums on the AY

The Problem: The AY Cannot Play Samples

The AY-3-8910, as we covered in Chapter 11, is a synthesiser. It generates square waves, noise, and envelope shapes. It has no sample playback capability, no DAC, no waveform RAM. Every sound it makes is built from those primitive sources in real time. If you want a realistic kick drum – the kind with a sharp transient punch followed by a resonant decay – the AY's noise generator and envelope can

approximate it, but the result sounds unmistakably synthetic. It lacks the weight of a real percussive hit.

But there is a back door.

Registers R8, R9, and R10 control the volume of channels A, B, and C. Each is a 4-bit value (0-15). If you write to a volume register once per frame, you get a static volume level. But what if you write to it thousands of times per frame? What if you treat the volume register as a crude 4-bit DAC and feed it successive sample values from a digitised recording?

You get PCM playback. Crude, noisy, 4-bit, but recognisable. The AY becomes a sample player – not by design, but by brute force.

The Cost: CPU Annihilation

Here is the problem. To play a digitised drum sample at any reasonable quality, you need to update the volume register at audio rates. A sample rate of 8 kHz means one update every 125 microseconds. At 3.5 MHz, 125 microseconds is approximately 437 T-states. That is tight but feasible – you can do useful work in the gaps between sample writes.

But 8 kHz sounds terrible. For a punchy kick drum, you want at least the perception of higher fidelity. And here the economics collapse. At higher effective sample rates, you need an interrupt or a tight polling loop that fires every 125-250 T-states. At that frequency, there is almost no CPU time left for anything else. While the drum sample is playing, the processor is a dedicated audio playback engine. Video generation, scripting, input handling – everything stops.

A typical kick drum sample lasts 20-40 milliseconds for the critical attack portion. At 50 Hz, that is 1-2 frames. During those frames, the CPU is gone.

n1k-o's Insight: The Hybrid Drum

n1k-o, the musician behind Eager's soundtrack, found the solution. The key observation: a drum sound has two distinct phases. The **attack** – the initial transient, the sharp "click" or "thud" that gives a kick drum its punch – is short, complex, and impossible to synthesise convincingly on the AY. But the **decay** – the resonant tail that follows – is a smooth volume falloff, exactly the kind of thing the AY's envelope generator handles naturally.

The hybrid approach: play the attack as a digital sample (consuming CPU time for 1-2 frames), then hand off to the AY's envelope generator for the decay (consuming zero CPU time, since the hardware does the work automatically). Digital attack plus AY decay equals a drum sound that has the realistic punch of a sample and the smooth tail of hardware synthesis.

In practice, the implementation works like this:

"'z80 id:ch12_n1k_o_s_insight_the_hybrid ; Play hybrid kick drum ; 1. Start digital sample playback for attack phase ; 2. When sample ends, configure AY envelope for decay

play_kick_drum: di ; disable interrupts – timing critical

```
; --- Digital attack phase ---
```

```
; Play ~800 samples at ~8kHz = ~100ms = ~2 frames
ld   hl, kick_sample      ; pointer to 4-bit sample data
ld   b, 0                 ; 256 samples per loop pass
ld   c, $FD               ; low byte of AY data port ($BFFD)

; Select volume register R8 (channel A)
ld   a, 8
ld   bc, $FFFD
out  (c), a               ; select R8
ld   c, $FD               ; prepare for $BFFD writes
```

.sample_loop: ld a, (hl) ; 7 T - load sample byte inc hl ; 6 T - advance pointer ld b, $BF ; 7 T - high byte of $BFFD out (c), a ; 12 T - write volume = sample value ; ... timing padding to hit target sample rate ... djnz .sample_loop ; 13 T (approx 45 T per sample)

```
; --- AY decay phase ---
; Configure envelope for smooth volume decay
; The AY takes over -- zero CPU cost from here

ld   a, R_ENV_LO
ld   e, 200               ; envelope period: moderate decay speed
call ay_write
ld   a, R_ENV_HI
ld   e, 0
call ay_write
ld   a, R_ENV_SHAPE
ld   e, $00               ; \___  single decay to silence
call ay_write
ld   a, R_VOL_A
ld   e, $10               ; switch channel A to envelope mode
call ay_write

ei
ret
```

kick_sample: ; 4-bit PCM data: attack portion of a kick drum ; Each byte = one sample, value 0-15 DB 0, 2, 8, 15, 14, 12, 15, 13 DB 10, 14, 11, 8, 12, 9, 6, 10 ; ... (typically 400-800 bytes for the full attack)

```
The sample data itself -- those 400-800 bytes of 4-bit PCM -- comes from a real drum recording

The result is convincing. On a chip that has no sample playback capability at all, you hear so

### The Frame Budget: Two Frames Per Hit

The frame cost is concrete: two frames per drum hit. During these frames, approximately 140,00
states (two full frame periods on Pentagon) are consumed by the sample playback loop. The CPU

Two frames at 50 Hz is 40 milliseconds. For a musical track with kick drums on the beat at 130

This is the architectural challenge that drives the rest of the chapter. How do you keep the v
```

```
> **Sidebar: MCC -- More Than 16 Levels from One AY**
>
> The single-channel approach above gives you 16 volume levels (4-bit DAC). But the AY has thr
>
> This is the MCC (Mixed Channel Covox) technique, documented by UnBEL!EVER (Born Dead #09, 19
bit sample values to three register values, you can synthesise approximately 108 distinct ampl
>
> The standard MCC playback loop runs at ~24 KHz (144 T per sample). A super-
fast variant by MOn5+Er (Born Dead #0G, 2000) uses SP as the sample data pointer (`POP HL` rea
>
> **The catch:** the MCC lookup table depends on the exact analogue output levels of each chan
3-8910 and YM2149F have different volume curves (the YM is closer to linear, the AY is more lo
level table was calculated for the YM2149F. On a different chip, the combined levels shift, an
>
> For demo work targeting a specific machine (say, Pentagon with YM2149F), MCC works well. For
hardware compatibility, the single-channel 4-bit approach is safer. And for fun experiments --
beat formulas (Appendix I) through an averaged MCC table -- the distortion is part of the char
>
> *Sources: UnBEL!EVER, "Воспроизведение оцифровок на AY (MCC)," Born Dead #09 (1999); MOn5+Er

---

## 12.2 Asynchronous Frame Generation

### The Naive Approach Fails

The simplest demo architecture is synchronous: generate one frame of the visual effect, wait f

Now add digital drums. The music engine signals: "play kick drum on the next beat." The sample

With one drum hit every 23 frames, the audience sees a brief freeze every half-
second. It is noticeable. It is ugly. It is unacceptable for a competition demo.

### Introspec's Solution: Stockpile Frames

Introspec's architecture in Eager decouples frame generation from frame display. The visual en

The mechanism is double-buffered attribute frames. Two pages of attribute data exist in memory

```text
Time ──────────────────────────────────────────────────────►

Display:   [Frame 1] [Frame 2] [Frame 3] [Frame 4] [Frame 5]
Generator: ──gen F2──|──gen F3──|──gen F4──|── DRUM ──|──gen F5──
                                           ↑          ↑
                                      drum starts  drum ends

During the drum hit, the display shows Frame 4 (already generated).
Frame 5 generation resumes immediately after the drum finishes.
```

But simple double buffering only gives you one frame of slack. If the drum consumes
```

two frames, you need to have generated two frames ahead. This is where Introspec's asynchronous generation truly diverges from simple double buffering: the engine can **stockpile** multiple frames in advance.

On the 128K Spectrum, memory banking provides the space. Attribute frames are small – 768 bytes each. A single 16KB memory page can hold roughly 20 attribute frames. The generator runs as fast as it can, writing frame after frame into the buffer. The display system reads from the buffer at a steady 50 Hz. When the generator is faster than real time (which it usually is, since attribute plasma is cheap), the buffer fills up. When a drum hit pauses generation, the display system draws down the buffer. As long as the buffer does not run dry, the audience sees smooth 50 Hz animation.

The Buffer Dynamics

Think of it as a producer-consumer problem, but on a machine with no concurrency.

The **producer** is the plasma/tunnel/zoomer effect generator. It produces attribute frames at a variable rate – sometimes faster than 50 Hz (when the calculation is simple and no drums are playing), sometimes zero (during drum playback).

The **consumer** is the display system, reading one frame per screen refresh at exactly 50 Hz.

The **buffer** sits between them, absorbing the difference.

The dynamics are straightforward:

- **Between drum hits:** The generator runs faster than the display. The buffer fills up. If it reaches capacity, the generator idles (or the engine advances the scripting state).
- **During a drum hit:** The generator stops. The display drains the buffer at 50 Hz. A two-frame drum hit consumes two buffered frames.
- **After a drum hit:** The generator resumes, running as fast as possible to refill the buffer before the next hit.

The critical constraint: **the buffer must never run dry during a drum hit.** If two drum hits occur in rapid succession – say, a kick-snare pattern two frames apart – the buffer needs at least four frames of reserve. Introspec's scripting engine manages this by knowing the music timeline in advance. When a dense drum passage approaches, the engine generates extra frames to pad the buffer. When a quiet passage follows, the buffer naturally fills.

The catch: if the drum pattern is too dense – too many hits too close together – the generator cannot keep up. The buffer runs dry, and the display repeats a frame. This is a hard constraint of the architecture, and it influenced n1k-o's composition. The music was written with knowledge of the engine's capacity: drum hits are spaced far enough apart that the generator can always recover. The musician and the coder designed together, each understanding the other's constraints.

12.3 The Scripting Engine

Why You Need a Script

By this point, the list of things that need coordination is long:

- The visual effect generator (which effect is active, what parameters it uses)
- The music player (which pattern is playing, when drums trigger)
- The frame buffer (how full it is, when to generate more)
- Transitions between effects (fade out one, fade in the next)
- The overall timeline (the demo runs for two minutes – what happens when)

You could hardcode all of this in a monolithic main loop. Some demos do. But Introspec chose a different path: a two-level scripting system that separates *what happens* from *when it happens*.

Outer Script: The Sequence of Effects

The outer script is a linear sequence of commands that control the overall structure of the demo. Think of it as a setlist for a concert:

```
; Outer script (conceptual, not exact syntax)
EFFECT  tunnel, params_set_1     ; start the tunnel effect
WAIT    200                       ; run for 200 frames (4 seconds)
EFFECT  zoomer, params_set_1     ; switch to chaos zoomer
WAIT    150                       ; 3 seconds
EFFECT  tunnel, params_set_2     ; tunnel again, different colours
WAIT    250                       ; 5 seconds
; ... and so on for the full demo
```

Each `EFFECT` command loads the generator function and its parameter block. Each `WAIT` tells the engine how many frames to run the current effect before advancing to the next command. Transitions between effects – crossfades, hard cuts, colour sweeps – are themselves scripted as effects.

Inner Script: Variations Within an Effect

Within a single effect, parameters change over time. The tunnel's plasma frequencies shift, the colour palette rotates, the zoom speed accelerates. These variations are controlled by the inner script – a per-effect sequence of parameter changes keyed to frame numbers:

```
; Inner script for tunnel effect (conceptual)
FRAME  0:   plasma_freq = 3, palette = warm
FRAME  50:  plasma_freq = 5                   ; frequency shift
FRAME  100: palette = cool                     ; colour change
FRAME  120: plasma_freq = 2, palette = hot     ; both change
```

The inner script runs independently of the outer script. When the outer script says "run tunnel for 200 frames," the inner script handles the visual evolution within those 200 frames.

kWORK: The Key Command

The most important command in the scripting system is what Introspec calls **kWORK**: “generate N frames, then show them independently of generation.” This single command is the bridge between the scripting system and the asynchronous architecture.

When the engine encounters `kWORK 8`, it:

1. Generates 8 frames of the current effect into the frame buffer.
2. Hands those frames to the display system.
3. While the display system shows them (over 8/50 = 160ms), the engine is free to do other work: process the next script command, prepare the next batch, or yield CPU time for drum playback.

This decoupling – generate now, display later – is the fundamental enabler for asynchronous operation. Without kWORK, the engine would be locked into a synchronous generate-display-generate-display cycle with no slack for drum interruptions.

In practice, the engine calls kWORK repeatedly, generating small batches of frames (4-8 at a time). Between batches, it checks whether a drum trigger is pending. If so, it lets the drum play, knowing that the display system has enough buffered frames to continue smoothly. After the drum finishes, it generates the next batch to replenish the buffer.

“‘z80 id:ch12_kwork_the_key_command ; Simplified engine loop (conceptual) engine_loop: ; Check if drum is pending ld a, (drum_pending) or a jr z, .no_drum call play_drum ; consumes 2 frames of CPU time xor a ld (drum_pending), a

.no_drum: ; Generate a batch of frames call generate_batch ; kWORK: produce N frames into buffer ; (generate_batch returns when batch is done)

```
; Check outer script for effect changes
call advance_script

jr   engine_loop


The beauty of this architecture is its simplicity at the macro level. The engine is a loop: ch

---

## 12.4 GABBA's Innovation: The Video Editor as Timeline Tool

In 2019, diver4d (of 4th Dimension) took first place at CAFe with GABBA, a gabber-
themed demo with punishingly tight audio-visual synchronisation. The sync was so precise that

The technical surprise was in the workflow, not the code.

### The Problem with Code-Based Sync

The traditional approach to music sync in ZX demos is to embed timing data in the code. You kn
test, and repeat. For a two-minute demo at 50 fps, that is 6,000 frames of potential sync poin

Introspec's Eager was built this way, and the development was gruelling. Every sync adjustment
```

diver4d's Answer: Luma Fusion

diver4d bypassed the code-edit-compile-test cycle entirely. He used **Luma Fusion**, an iOS vi

The workflow:

1. **n1k-o composed the gabber track** with frame-level structural markers. The musician and c
accurate map of the entire track.

2. **diver4d recorded each visual effect** running at 50 fps in an emulator and exported the r

3. **In Luma Fusion**, he arranged the video clips on a 50 fps timeline alongside the audio tr

4. **Once the timing was right in the editor**, he extracted the frame numbers for each transi

The insight is straightforward: use the right tool for the job. A video editor is purpose-
built for frame-level multimedia synchronisation. Z80 assembly is not. By doing the creative s

What This Changes

The immediate benefit is speed. Adjusting sync timing in a video editor takes seconds. Adjusti
based iteration -- the feedback loop is too slow.

The limitation is that this workflow works best for demos where the timing is fixed -- where t
based approach. But for the overwhelming majority of ZX demos, which are linear fixed-
timeline productions, the video editor workflow is superior.

GABBA demonstrated that demoscene production tools do not have to be retro. The Z80 code is fr

---

12.5 Z80 Threading: A Different Path

Robus, writing in Hype in 2015, presented a technique that attacks the concurrency problem fro

The Problem, Restated

The fundamental tension in a demo engine is that multiple tasks need CPU time in the same fram

What if the Z80 could run two tasks simultaneously?

IM2-Based Context Switching

It can, after a fashion. The Z80's IM2 interrupt provides a natural context switch point. Ever

Robus's `SwitchThread` procedure does exactly this:

```z80 id:ch12_im2_based_context_switching
; SwitchThread: save current thread, resume next thread
; Called from within the IM2 interrupt handler
```

```
SwitchThread:
    ; Save current thread's stack pointer
    ld   (thread_sp_save), sp

    ; Save current memory page configuration
    ld   a, (current_7ffd)
    ld   (thread_page_save), a

    ; Load next thread's state
    ld   a, (next_thread_page)
    ld   (current_7ffd), a
    ld   bc, $7FFD
    out  (c), a               ; switch memory page

    ld   sp, (next_thread_sp)  ; switch stack pointer

    ; Execution continues in the next thread's context
    ; (it was previously suspended at this same point)
    ret
```

Each thread gets its own **128-byte stack** and a **dedicated memory page** (one of the 128K Spectrum's eight 16KB banks). The stack is small but sufficient – Z80 code rarely nests deeply. The dedicated memory page gives each thread its own workspace without interfering with the other.

How It Works in Practice

In Robus's WAYHACK demo, two threads run concurrently:

- **Thread 1:** Calculates the visual effect (a dungeon-crawling perspective renderer).
- **Thread 2:** Renders scrolling text along the bottom of the screen.

Neither thread knows about the other. Each runs in its own memory page with its own stack. Every frame, the IM2 interrupt fires and `SwitchThread` alternates between them. Thread 1 gets one frame of CPU time, then Thread 2 gets one frame, and so on.

The result: the text scroller runs at a steady 25 Hz (every other frame), and the visual effect runs at 25 Hz. Neither task needs to be aware of the other's existence. No cooperative scheduling, no yield points, no manual interleaving. The interrupt handles everything.

The Threading Model

The model is simple:

```
Frame 1: Interrupt → save Thread 2 → restore Thread 1 → Thread 1 runs
Frame 2: Interrupt → save Thread 1 → restore Thread 2 → Thread 2 runs
Frame 3: Interrupt → save Thread 2 → restore Thread 1 → Thread 1 runs
...
```

Each thread sees a consistent world: its registers, its stack, its memory page. The switch happens at a fixed point (the interrupt), so there are no race conditions on

shared data. If the threads need to communicate (e.g., Thread 1 signals Thread 2 to change the text), they do so through a shared memory location that both threads can access – a simple flag or mailbox.

Practical Considerations

Robus's own assessment is characteristically honest: **"Honest multithreading rarely requires more than two threads"** on the Z80. The overhead of context switching (saving and restoring SP plus a memory page switch) is modest – perhaps 100 T-states – but each additional thread halves the available CPU time per thread. With two threads, each gets 25 Hz. With three, each gets roughly 16.7 Hz. On a machine where visual smoothness demands close to 50 Hz, two threads is the practical limit.

The threading approach is orthogonal to Introspec's asynchronous buffering approach. You could combine them: one thread generates effect frames into a buffer while the other handles music and drum playback. In practice, this combination is rare – the two techniques solve the same problem (interleaving CPU-hungry tasks) through different mechanisms, and most demo coders choose one or the other based on the specific demands of their production.

Threading works best when two tasks are truly independent and neither needs more than 25 Hz. The asynchronous buffer approach works best when one task (visuals) needs 50 Hz and the other (drums) needs unpredictable bursts. For Eager's architecture, where visual smoothness was paramount and drum timing was dictated by the music, the buffer approach won. For WAYHACK's architecture, where two steady-state tasks ran in parallel, threading won.

12.6 Practical: A Minimal Scripted Demo Engine

Let us build a minimal demo engine that ties together the concepts from this chapter. The goal is not Eager-level sophistication – it is a skeleton that demonstrates the architecture.

What We Build

- **Three simple effects:** plasma (attribute-based, from Chapter 9), colour bars (horizontal attribute stripes), and a text scroller.
- **AY music** playing via IM2 interrupt (using a .pt3 player, as described in Chapter 11).
- **A digital kick drum sample** that plays on the beat, stealing 2 frames of CPU.
- **A simple timeline script** that switches between effects at defined points.
- **Double-buffered attributes** to absorb the drum hit pauses.

The Memory Map

```
$6000-$7FFF   Engine code + effect routines
$8000-$9FFF   Music player + song data
$A000-$AFFF   Sine tables, colour maps, sample data
$B000-$BFFF   Frame ring buffer (attribute frames)
```

```
$C000-$DFFF   Shadow screen (second display page)
$E000-$FFFF   Stack + IM2 vector table + workspace

Bank 0-3:     Not used (available for larger effects)
Bank 5:       Normal screen ($4000-$5AFF display)
Bank 7:       Shadow screen ($C000-$DAFF display)
```

The Timeline Script

“‘z80 id:ch12_the_timeline_script ; Timeline script: sequence of (effect_id, duration_frames, param_ptr) timeline: DB EFFECT_PLASMA, 0, 150 ; plasma for 150 frames (3 sec) DW plasma_params_1 DB EFFECT_BARS, 0, 100 ; colour bars for 100 frames (2 sec) DW bars_params_1 DB EFFECT_SCROLLER, 0, 200 ; text scroller for 200 frames (4 sec) DW scroller_params_1 DB EFFECT_PLASMA, 0, 150 ; plasma again, different params DW plasma_params_2 DB $FF ; end marker: loop from start

EFFECT_PLASMA EQU 0 EFFECT_BARS EQU 1 EFFECT_SCROLLER EQU 2

The Main Engine Loop

```z80 id:ch12_the_main_engine_loop
; Main engine loop
; Assumes IM2 is set up and music player runs in the ISR

engine_init:
    ; Set up display: fill pixel memory with checkerboard
    call fill_checkerboard

    ; Initialise ring buffer
    xor  a
    ld   (buf_write_idx), a
    ld   (buf_read_idx), a
    ld   (buf_count), a

    ; Load first effect from timeline
    ld   hl, timeline
    ld   (script_ptr), hl
    call load_next_effect

engine_main:
    ; === Step 1: Check for drum trigger ===
    ld   a, (drum_pending)
    or   a
    jr   z, .no_drum

    ; Play the drum -- this consumes ~2 frames
    call play_kick_drum
    xor  a
    ld   (drum_pending), a
    jr   .after_drum
```

```
.no_drum:
    ; === Step 2: Generate a frame into the buffer ===
    ld   a, (buf_count)
    cp   BUF_CAPACITY         ; buffer full?
    jr   nc, .buffer_full

    ; Generate one frame of the current effect
    call generate_frame       ; writes 768 bytes to ring buffer

    ; Advance buffer write pointer
    ld   a, (buf_write_idx)
    inc  a
    cp   BUF_CAPACITY
    jr   nz, .no_wrap_w
    xor  a
.no_wrap_w:
    ld   (buf_write_idx), a
    ld   a, (buf_count)
    inc  a
    ld   (buf_count), a

.buffer_full:
.after_drum:
    ; === Step 3: Advance timeline ===
    ld   hl, (frame_counter)
    inc  hl
    ld   (frame_counter), hl

    ; Check if current effect duration has elapsed
    ld   de, (effect_duration)
    or   a
    sbc  hl, de
    jr   c, .effect_continues

    ; Load next effect from timeline
    call load_next_effect
    ld   hl, 0
    ld   (frame_counter), hl

.effect_continues:
    ; === Step 4: Wait if we are ahead of display ===
    halt                      ; sync to frame boundary

    jr   engine_main
```

The Display ISR

“‘z80 id:ch12_the_display_isr ; IM2 interrupt handler: runs every frame (50 Hz) frame_isr: push af push bc push de push hl

```
; Play music (updates AY registers)
```

```
call music_play

; Check if music engine signals a drum hit
ld   a, (music_drum_flag)
or   a
jr   z, .no_drum_signal
xor  a
ld   (music_drum_flag), a
ld   a, 1
ld   (drum_pending), a    ; signal main loop
```

.no_drum_signal:

```
; Display next frame from ring buffer
ld   a, (buf_count)
or   a
jr   z, .no_frame         ; buffer empty, keep current frame

; Copy buffered attributes to display page
call copy_buf_to_screen

; Advance read pointer
ld   a, (buf_read_idx)
inc  a
cp   BUF_CAPACITY
jr   nz, .no_wrap_r
xor  a
```

.no_wrap_r: ld (buf_read_idx), a ld a, (buf_count) dec a ld (buf_count), a

.no_frame: pop hl pop de pop bc pop af ei reti

BUF_CAPACITY EQU 8 ; 8 frames of buffer (8 x 768 = 6,144 bytes)

The Effect Generator Dispatch

```z80 id:ch12_the_effect_generator_dispatch
; Generate one frame of the current effect
; Writes attribute data to the ring buffer
generate_frame:
    ld   a, (current_effect)
    or   a
    jr   z, .do_plasma
    cp   1
    jr   z, .do_bars
    cp   2
    jr   z, .do_scroller
    ret

.do_plasma:
    call calc_plasma          ; from Chapter 9 -- writes 768 bytes
    ret
.do_bars:
```

```
    call calc_colour_bars     ; horizontal attribute stripes
    ret
.do_scroller:
    call calc_text_scroll     ; text rendering into attributes
    ret
```

Observations

This skeleton is deliberately simple. A production engine would add:

- **Inner scripts** for parameter variation within each effect.
- **Transition effects** (crossfades between two attribute buffers).
- **Multiple drum sounds** (kick, snare, hi-hat), each with its own sample data.
- **Buffer level monitoring** so the generator can prioritise catching up after dense drum passages.
- **Memory banking** to store more frames and support larger effect data.

But even in this minimal form, the architecture demonstrates the key principles:

1. **Decoupled generation and display.** The generator and the display ISR communicate only through the ring buffer. Neither knows or cares about the other's timing.

2. **Drum hits are absorbed by the buffer.** When `play_kick_drum` consumes two frames, the display ISR continues showing buffered frames. The audience sees no stutter.

3. **The script drives the timeline.** Adding a new effect or changing the sequence means editing the `timeline` data table, not restructuring the engine code.

4. **The music player runs in the ISR.** It updates AY registers every frame regardless of what the main loop is doing. The only interaction is the `drum_pending` flag – a one-byte mailbox between the ISR and the main loop.

This is the architecture of a demo. Not the effects, not the music, not the art – the *plumbing* that makes all of those work together. It is the least visible part of a demo and the hardest to get right. Introspec spent ten weeks on Eager, and the architecture consumed more of that time than any single effect.

12.7 Practical Exercises

Exercise 1: Basic Engine. Implement the skeleton above with a single effect (plasma from Chapter 9) and no drum samples. Verify that the ring buffer works correctly: the display shows smooth animation while the generator runs at its natural speed.

Exercise 2: Add the Drum. Record (or synthesise) a 4-bit kick drum sample (400-800 bytes). Add the `play_kick_drum` routine and trigger it every 25 frames. Verify that the display remains smooth during drum playback. What is the maximum drum rate before the buffer runs dry?

Exercise 3: Multi-Effect Timeline. Add a second effect (colour bars or text scroller). Write a timeline script that switches between effects every 3-4 seconds. Verify that transitions happen at the correct frame.

Exercise 4: Sync to Music. Load a short .pt3 tune and modify the player to set `music_drum_flag` when a particular pattern event occurs (e.g., a note on channel C below a certain pitch). Now the drums are driven by the music, not by a fixed frame counter. This is real music sync.

Exercise 5: Video Editor Workflow. Record your running demo in an emulator at 50 fps. Import the recording into a video editor (any editor that supports frame-level editing). Adjust the timeline script frame numbers based on what you see in the editor. Experience the difference in iteration speed compared to code-only sync.

Summary

This chapter was not about any single effect or technique. It was about architecture – the invisible structure that lets a demo exist as a coherent, synchronised, two-minute performance rather than a collection of disjointed screens.

The core problems are universal. Every demo engine must answer: How do I share the CPU between audio and video? How do I keep the display smooth when the audio steals processing time? How do I sequence effects and synchronise them to music? How do I manage the timeline of a multi-minute production?

The solutions we examined are complementary:

- **Digital drums** (n1k-o/Introspec) exploit the AY's volume registers as a crude DAC, blending digital samples with hardware synthesis to produce percussion that transcends the chip's designed capabilities.
- **Asynchronous frame generation** (Introspec) decouples video production from display through a ring buffer, absorbing the CPU bursts consumed by drum playback.
- **Scripted timelines** (Introspec) separate the *what* and *when* of a demo from the *how*, making it possible to design and adjust a two-minute production without restructuring the engine.
- **Video editor sync** (diver4d) moves the creative timing work to a tool purpose-built for it, dramatically accelerating the sync iteration cycle.
- **Z80 threading** (Robus) provides genuine concurrency for tasks that are independent and steady-state, at the cost of halving the frame rate for each task.

With this chapter, we close the circle on the demoscene section of the book. We have built effects (Parts I-II), made sound (Chapter 11), and now wired everything into a running engine. The reader who has followed from Chapter 1 has a complete picture: from T-state counting to a synchronised, scripted demo with digital drums.

In the next chapter, we shift gears entirely. Part IV takes us into size-coding – the art of fitting an entire production into 256 bytes. The architecture goes from "how do I manage a ring buffer?" to "how do I make every single byte do double duty?" The constraints tighten by three orders of magnitude, and the thinking changes to match.

Sources: Introspec, “Making of Eager,” Hype, 2015 (hype.retroscene.org/blog/demo/261.html); Introspec, file_id.diz from Eager (to live), 3BM Open Air 2015; diver4d, “Making of GABBA,” Hype, 2019 (hype.retroscene.org/blog/demo/948.html); Robus, “Threads on Z80,” Hype, 2015 (hype.retroscene.org/blog/dev/271.html); Eager source code excerpts courtesy of Introspec (Life on Mars)

Chapter 13: The Craft of Size-Coding

> "It was like playing puzzle-like games – constant reshuffling of code to find shorter encodings." – UriS, on writing NHBF (2025)

There is a category of demoscene competition where the constraint is not time but *space*. Your entire program – the code that draws the screen, produces the sound, handles the frame loop, holds whatever data it needs – must fit in 256 bytes. Or 512. Or 1K, or 4K, or 8K. Not a byte more. The file is measured, and if it is 257 bytes, it is disqualified.

These are **size-coding** competitions, and they produce some of the most remarkable work on the ZX Spectrum scene. A 256-byte intro that fills the screen with animated patterns and plays a recognisable melody is a form of compression so extreme it is hard to believe until you read the code. The gap between what the audience sees and the file size that produces it – that gap is the art.

This chapter is about the mindset, the techniques, and the specific tricks that make size-coding possible.

13.1 What is Size-Coding?

Demo competitions typically offer several size-limited categories:

Category	Size limit	What fits
256 bytes	256	One tight effect, maybe simple sound
512 bytes	512	An effect with basic music or two simple effects
1K intro	1,024	Multiple effects, proper music, transitions
4K intro	4,096	A short demo with several parts
8K intro	8,192	A polished mini-demo

The limits are absolute. The file is measured in bytes, and there is no negotiation.

What makes size-coding fascinating is that it inverts the normal optimisation hierarchy. In the cycle-counted world of demoscene effects, you optimise for *speed* – unrolling loops, duplicating data, generating code, all trading space for time. Size-coding inverts this. Speed does not matter. Readability does not matter. The only question is: can you make it one byte shorter?

UriS, who wrote the 256-byte intro NHBF for Chaos Constructions 2025, described the process as "playing puzzle-like games." The description is exact. Size-coding is a puzzle where the pieces are Z80 instructions, the board is 256 bytes of RAM, and the best solutions involve moves that solve multiple problems simultaneously.

The mindset shift:

- **Every byte is precious.** A 3-byte instruction where a 2-byte one suffices is 0.4% of your entire program. At 256 bytes, one byte saved is like saving 250 bytes in a 64K program.
- **Code and data overlap.** The same bytes that execute as instructions can serve as data. The Z80 does not know the difference – only the program counter's path through memory distinguishes code from data.
- **Instruction choice is driven by size, not speed.** `RST $10` costs 1 byte. `CALL $0010` does the same thing in 3 bytes. In a normal demo you would never notice. In 256 bytes, those 2 bytes are the difference between having sound or not.
- **Initial state is free data.** After boot, registers have known values. Memory at certain addresses contains known data. A size-coder exploits every bit of this free state.
- **Self-modifying code is not a trick – it is a necessity.** When you cannot afford a separate variable, you modify an instruction's operand in place.

The Z80 Size-Coder's Toolkit

Some tricks recur so often in size-coded intros that they form a shared vocabulary. Knowing these by heart – their byte costs, their side effects – is the prerequisite to serious size-coding.

Register initialization assumptions. When a Spectrum program launches from BASIC (via `RANDOMIZE USR`), the CPU state is not random. After `CLEAR` and before the USR call, A is typically 0, BC holds the USR address, DE and HL have known values from the BASIC interpreter, the stack pointer sits at the CLEAR address, and interrupts are enabled. Many of these are stable enough to rely on. If your program needs A = 0 at the start, do not write `XOR A` – it is already zero. If you need a 16-bit counter starting at 0, check whether DE or HL already holds 0 or a useful value. One byte saved here, two bytes saved there – these add up to the difference between 260 bytes and 256.

The system variables area ($5C00-$5CB5) is another source of free data. The BASIC interpreter maintains over 100 bytes of state at known addresses. If you need the value 2, you might find it at the address holding the current stream number. If you need $FF, several system variable fields contain it. Reading from a fixed address costs 3 bytes (`LD A, (nn)`), but if it replaces a 2-byte load *plus* some computation, you come out ahead.

DJNZ as a short backwards jump. `DJNZ label` is 2 bytes, same as `JR label` – but it also decrements B. If B is nonzero and you need a backwards jump, `DJNZ` does both for free. Even when you do not care about B's decrement, `DJNZ` is still 2 bytes, the same cost as `JR`. But if B happens to reach zero at the exact moment you want to fall through, you have merged a loop counter and a branch into a single instruction. Size-coders routinely structure loops so that B's natural countdown aligns with the exit condition.

RST as a 1-byte CALL. The Z80 reserves eight restart addresses: $00, $08, $10, $18, $20, $28, $30, $38. `RST n` pushes the return address and jumps to the target – the same as `CALL n` – but in 1 byte instead of 3. On the Spectrum, the ROM places useful routines at several of these addresses:

- `RST $10` – print a character (ROM routine at $0010)
- `RST $20` – collect next character from BASIC (less useful for demos)
- `RST $28` – enter the floating-point calculator (useful for math)
- `RST $38` – the maskable interrupt handler (IM 1 jumps here)

In a normal demo, these ROM routines are too slow to call in a tight loop. In a 256-byte intro, saving 2 bytes per call is worth the speed penalty. If your program calls `RST $10` six times to print characters, that is 12 bytes saved over six `CALL $0010` instructions. Twelve bytes is nearly 5% of 256.

Overlapping instructions. The Z80 decodes instructions byte by byte, with no alignment requirements. If you jump into the middle of a multi-byte instruction, the CPU decodes fresh from that point. This means you can hide one instruction inside another:

```
z80 id:ch13_the_z80_size_coder_s_toolkit     ld   a, $AF              ; opcode
$3E, operand $AF                               ; BUT: $AF is XOR A
```

If the CPU executes from the start, it sees `LD A, $AF` (2 bytes). If another code path jumps to the second byte, it sees `XOR A` (1 byte). One byte serves two purposes. The technique is fragile – it demands perfect control of all execution paths – but in competition code, fragility is acceptable.

A common pattern: the byte `$18` is `JR d` (relative jump). If you need the value $18 as data *and* need a branch at that location, the same byte does both. The operand that follows is both the jump offset and (from another perspective) the next piece of data.

Abusing flag state. Every arithmetic and logical instruction sets flags. Size-coders memorise which flags each instruction affects and exploit the results instead of computing them separately. After `DEC B`, the zero flag tells you whether B hit zero – no `CP 0` needed. After `ADD A, n`, the carry flag tells you whether the result overflowed past 255. After `AND mask`, the zero flag tells you whether any masked bits were set.

The deepest flag trick is `SBC A, A`: if carry is set, A becomes $FF; if carry is clear, A becomes $00. One byte, no branch, a full bitmask from a flag. Compare this to the branching alternative:

“‘z80 id:ch13_the_z80_size_coder_s_toolkit_2 ; With branching: 6 bytes jr nc, .zero ; 2 ld a, $FF ; 2 jr .done ; 2 .zero: xor a ; 1 .done:

```
; With SBC A,A: 1 byte
sbc  a, a                ; 1 — carry -> $FF, no carry -> $00


Five bytes saved. In a 256-byte intro, that is two percent of the entire program.

---

## 13.2 Anatomy of a 256-Byte Intro: NHBF
```

NHBF (No Heart Beats Forever) was created by UriS for Chaos Constructions 2025, inspired b
wave power chords with random pentatonic melody notes -- all in 256 bytes.

The Music

At 256 bytes, you cannot include a tracker player or note tables. NHBF drives the AY chip dire
write instructions -- the same bytes that form the `LD A, n` operand *are* the musical note. T
random generator (typically `LD A, R` -- read the refresh register -- followed by AND to mask

The Visual

Printing text through the ROM -- `RST $10` outputs a character for 1 byte per call -- is the c
character string costs 40 bytes (character codes + RST calls). Size-coders look for ways to co

The Puzzle: Finding Overlaps

UriS describes the core process as constant reshuffling. You write a first version at 300 byte
clearing routine uses LDIR, which decrements BC to zero. Arrange the code so the next section

Every instruction produces side effects -- register values, flag states, memory contents -- an

Art-Top's Discovery

During development, Art-Top noticed something remarkable: the register values left over from t
clearing routine happened to match the exact length needed for the text string. Not planned. U

This kind of serendipitous overlap is the heart of 256-byte coding. You cannot plan for it. Yo

The Byte Budget

When working at 256 bytes, a rough budget helps you plan before writing a single instruction.
byte intro with both visuals and sound:

Component	Bytes	Notes
Pixel fill (dither/clear)	18-25	LD HL, LDIR or a compact fill loop
AY initialisation	16-22	Mixer, volume, initial tone — via port writes
Main loop frame sync	1	HALT
AY tone update per frame	10-14	Select register, write tone period
Visual effect core	30-50	The inner loop that computes and writes attributes
Outer loop / row control	8-12	Row counter, column counter, branches
Frame counter update (SMC)	6-8	Read, increment, write back into instruction
Loop back to main	2	JR main_loop
Total framework	**~91-134**	Before any effect-specific code

That leaves 122-165 bytes for the actual creative content -- the visual formula, data tables,
coders fight so hard for every byte in the scaffolding: each byte saved in the framework is a

Look at the companion example `intro256.a80`. Its pixel fill loop uses 18 bytes. The AY setup

Key Techniques at 256 Bytes

1. Use initial register and memory state. After a standard tape load, registers hold known
$5CB5) contains known values. Screen memory is clear after CLS. Every known value you exploit
3 bytes.

2. Overlap code and data. The byte $3E is the opcode for `LD A, n` and also the value 62 -

3. Choose instructions for size.

Large encoding	Small encoding	Savings
`CALL $0010` (3 bytes)	`RST $10` (1 byte)	2 bytes
`JP label` (3 bytes)	`JR label` (2 bytes)	1 byte
`LD A, 0` (2 bytes)	`XOR A` (1 byte)	1 byte
`CP 0` (2 bytes)	`OR A` (1 byte)	1 byte

The RST instructions are critical. `RST n` is a 1-byte CALL to one of eight addresses ($00, $0

Every JP in a 256-byte intro should be a JR -- the whole program fits within the -
128..+127 range.

4. Self-modifying code to reuse sequences. Need a subroutine to operate on two different a

5. Mathematical relationships between constants. If your music needs tone period 200 and y

---

13.3 Famous 256-Byte Intros: What Made Them Clever

The ZX Spectrum 256-byte category has a rich history. Studying winning entries reveals what ki

Attribute-based effects dominate. The reason is arithmetic: the Spectrum's attribute area
20 bytes. Pixel-level effects require addressing 6,144 bytes of interleaved screen memory -- m
byte intros work in attribute space: colour plasmas, interference patterns, gradient animation
time dither fill, or is left with whatever the ROM puts there.

Generative sound beats sequenced sound. A note table for a melody costs bytes -- even a si
note sequence is 8 bytes, plus the indexing logic. At 256 bytes, the winning strategy is to de
random source, then mask it to a pentatonic range. The sound will not be a composition, but it

The ROM is your library. Every byte of the Spectrum's 16K ROM is available and does not co
point calculator, which can compute sine, cosine, and square roots -- operations that would co
byte intro running at 50fps with a simple effect, you often have cycles to spare.

The entries that win are the ones that look impossible at their size. Judges and audiences
byte intro with a smooth colour plasma and a recognisable melody generates more applause than
based interference patterns are perfect: visually intricate, mathematically trivial. Colour cy
loop code if the formula is chosen carefully.

```
## 13.4 The LPRINT Trick

In 2015, diver4d published "Secrets of LPRINT" on Hype, documenting a technique older than the

### How It Works

The system variable at address 23681 ($5C81) controls where BASIC's output routines direct dat

```basic
10 POKE 23681,64: LPRINT "HELLO"
```

That single POKE redirects the printer channel to $4000 – the start of screen memory.

The Transposition Effect

The visual result is not just text on screen – it is *transposed* text. The Spectrum's screen memory is interleaved (Chapter 2), but the printer driver writes sequentially. Data lands in screen memory according to the driver's linear logic but *displays* according to the interleaved layout. The result cycles through 8 visual states as it progresses through the screen thirds – a cascade of data that builds in horizontal bands, shifting and recombining.

With different character data – graphical characters, UDGs, or carefully chosen ASCII sequences – the transposition produces striking visual patterns. The LPRINT statement handles all screen addressing, character rendering, and cursor advancement. Your program provides only the data.

From Pirate Loaders to Demo Art

diver4d traced the trick to pirated cassette loaders. Pirates adding custom loading screens needed visual effects in very few BASIC bytes – LPRINT was ideal. The technique fell out of use as the scene moved to machine code.

But in 2011, JtN and 4D released **BBB**, a demo that deliberately returned to LPRINT as an artistic statement. The old pirate-loader trick, framed with intention, became demo art. The constraint – BASIC, a printer redirect hack, no machine code – became the medium.

Why It Matters for Size-Coding

LPRINT achieves complex screen output for almost zero bytes of your own code. The ROM does the heavy lifting. Your contribution: a POKE to redirect output, data to print, and `RST $10` (or LPRINT) to trigger it. You leverage the Spectrum's 16K ROM as a "free" screen-output engine – code that does not count against your size limit.

13.5 512-Byte Intros: Room to Breathe

Doubling from 256 to 512 bytes is not twice as much – it is qualitatively different. At 256, you fight for every instruction and sound is minimal. At 512, you can have
```

a proper effect *and* proper sound, or two effects with a transition.

What Each Size Tier Enables

The jump between size categories is not linear. Each doubling opens qualitative new possibilities:

256 bytes is one effect and maybe primitive sound. You cannot afford a data table longer than about 16 bytes. Every variable lives in a register or in the instruction stream (self-modifying code). Text output is limited to a few characters. You have room for one nested loop with 2-3 arithmetic operations in the inner body. The visual will be attribute-based, generated purely from arithmetic. Sound, if present, is a tone sweep or random notes.

512 bytes lets you add a sine table (32-64 bytes), a real AY music engine (melody + bass on two channels), or a second visual effect with a transition. You can afford a proper frame-counted state machine that switches between two parts. Self-modifying code becomes structural rather than desperate. You might even have room for a short text string (10-20 characters) displayed with `RST $10`.

1K (1,024 bytes) is a different world. You can have a tracker-style music player with a compressed pattern (one channel with a 32-step loop takes about 80-120 bytes including the player). Multiple effects with transitions become standard. Pixel-level effects – simple plasma in pixel space, scrolling text, raster bars – become feasible because you can afford the screen memory address calculation. You can include a 256-byte sine table, or generate one at startup and keep it in a buffer. At 1K, the constraint still shapes every decision, but the decisions are about *which features to include*, not about *which instructions you can afford*.

4K and 8K intros approach the territory of short demos. At 4K, compression becomes viable and you can fit multi-effect compositions with music – a qualitative leap covered in Section 13.6. An 8K intro is a polished mini-demo where the constraint is more about data compression than instruction-level size tricks. The techniques from this chapter still apply, but the focus shifts from "can I save one byte?" to "can I compress this data stream?"

The sweet spot for learning size-coding is 256 bytes. At that size, every technique in this chapter is mandatory. At 512, you have enough room to choose. At 1K, the size-coding mindset helps but does not dominate.

Common 512-Byte Patterns

Plasma via sine table sums. The sine table is the expensive part. A full 256-byte table consumes half your budget. Solutions: a 64-entry quarter-wave table mirrored at runtime (saves 192 bytes), or generate the table at startup using the parabolic approximation from Chapter 4 (~20 bytes of code instead of 256 bytes of data).

Tunnel via angle/distance lookup. At 512 bytes, you compute angle and distance on the fly using rough approximations. Lower visual quality than the Eager tunnel (Chapter 9), but recognisably a tunnel.

Fire via cellular automaton. Each cell averages its neighbours below, minus decay. A few instructions per pixel, convincing animation, and at 512 bytes you can add attributes for colour *and* a beeper sound.

Self-Modifying Tricks

Self-modification becomes structural at 512 bytes. Embed the frame counter *inside* an instruction:

```
z80 id:ch13_self_modifying_tricks frame_ld:     ld   a, 0              ; this 0
is the frame counter     inc  a     ld   (frame_ld + 1), a  ; update the counter
in place
```

No separate variable. The counter lives in the instruction stream.

Patch jump offsets to switch between effects:

```
z80 id:ch13_self_modifying_tricks_2 effect_jump:     jr   effect_1              ;
this offset gets patched     ; ... effect_1:     ; render effect 1, then:     ld
a, effect_2 - effect_jump - 2     ld   (effect_jump + 1), a   ; next frame jumps
to effect 2
```

The ORG Trick

Choose your program's ORG address so that address bytes themselves are useful data. Place code at $4000 and every JR/DJNZ targeting labels near the start generates small offset bytes – usable as loop counters, colour values, or AY register numbers. If your effect needs $40 (the high byte of screen memory) as a constant, place code at an address where $40 appears naturally in an address operand. The *encoding of the code itself* provides data you need elsewhere.

This is the deepest level of the size-coding puzzle.

13.6 4K Intros: The Mini-Demo

4096 bytes is where size-coding transitions from "one trick" to "mini-demo." At 256 bytes, you have room for a single effect and maybe primitive sound. At 512 or 1K, you can have a proper effect with music. At 4K, you can have multiple effects, transitions between them, a full soundtrack, and a coherent narrative arc. The difference between 1K and 4K is qualitative, not just quantitative – it is the difference between "clever trick" and "tiny production."

Compression Becomes Viable

The single biggest change at 4K is that data compression pays for itself. A good Z80 decompressor – ZX0, Exomizer, or similar – costs roughly 150-200 bytes of code. At 256 or 512 bytes, that overhead is catastrophic. At 4K, it is less than 5% of your budget, and the return is enormous: a 4K intro might contain 6-8K of uncompressed code and data, packed down to fit the limit. Your actual working space nearly doubles.

The workflow becomes a feedback loop: write code, assemble to a raw binary, compress with ZX0, check the output size, iterate. The number that matters is no longer the assembled size – it is the *compressed* size. This changes your optimisation strategy. You are no longer counting individual instruction bytes. You are thinking about what compresses well.

Code with repetitive patterns compresses better than code with high entropy. A table of sine values compresses well (smooth, predictable). A table of random bytes does not. Effect code that reuses similar instruction sequences across routines compresses better than code where every routine has a unique structure. This is a subtle shift: you optimise not just for *small code* but for *compressible code*.

Music Fits

At 256 bytes, sound is a luxury – a tone sweep or random pentatonic notes. At 4K, you can have a real soundtrack. A tiny AY player engine – something like Beepola's output or a custom minimal tracker – takes 200-400 bytes. Add 500-1000 bytes of pattern data (compressed) and you have a full three-channel AY composition with melody, bass, and drums. These numbers compress well because music pattern data is highly repetitive.

The impact on the audience is disproportionate. Sound transforms a size-coding entry from a visual curiosity into an *experience*. At compo screenings, intros with music score dramatically higher than silent ones of equal visual quality. If you have 4K to work with and you are not including music, you are leaving points on the table.

Multi-Effect Structure

Unlike 256 bytes where you are locked into a single visual, 4K gives you room for 2-4 distinct effects with transitions. The structural framework is lightweight: a scene table mapping effect pointers to durations costs perhaps 30 bytes:

"'z80 id:ch13_multi_effect_structure scene_table: DW effect_plasma ; pointer to effect routine DB 150 ; duration in frames (3 seconds at 50fps) DW effect_tunnel DB 200 DW effect_scroller DB 250 DB 0 ; end marker

scene_runner: ld hl, scene_table .next_scene: ld e, (hl) inc hl ld d, (hl) ; DE = effect routine address inc hl ld a, (hl) ; A = duration or a ret z ; end of table inc hl push hl ; save table pointer ld b, a ; B = frame counter .frame_loop: push bc push de call .call_effect pop de pop bc halt ; wait for vsync djnz .frame_loop pop hl jr .next_scene

.call_effect: push de ret ; jump to DE via push+ret trick

```
Each individual effect might run 500-1000 bytes of code. At 4K compressed, you can afford thre

### GOA4K, inal, and Megademica

**GOA4K** by Exploder^XTM is a landmark ZX Spectrum 128K 4K intro that demonstrates what is ac
size demo, compressed down to a size you could fit in a single disk sector.

The story does not end there. **SerzhSoft** took GOA4K and remade it as **inal** -- a 48K-
only version in just 2980 bytes. The same visual impact, on a more constrained machine, in few
coding community at work: one coder sets a bar, another clears it from a harder starting posit

SerzhSoft went on to win the 4K intro compo at **Revision 2019** with **Megademica** -- compet
specific category, but against all platforms at the world's largest demoscene event. A ZX Spec
coding enables: from local scene technique to global recognition.
```

```
Studying entries like these reveals a pattern: the best 4K intros choose effects that are visu
test-iterate cycle.

### The 4K Tradeoffs

Working at 4K introduces tradeoffs that do not exist at smaller sizes:

**Compression ratio drives effect choice.** Not all effects compress equally. A plasma that re
random dithering effect where every pixel is computed from a different formula produces high-
entropy code that barely compresses at all. At 4K, you choose effects partly on their visual m

**Boot time is visible.** Decompression takes real time -- typically 1-3 seconds on a 3.5MHz Z
frame title screen. The decompressor itself runs from a small uncompressed stub at the start o

**You optimise for packed size, not runtime speed.** In a 256-byte intro, the same code that r

**Counting packed bytes.** The build process gains a compression step. Assemble to binary, com

```sh
sjasmplus --nologo --raw=build/intro4k.bin intro4k.a80
zx0 build/intro4k.bin build/intro4k.zx0
ls -l build/intro4k.zx0    # this is the number that must be <= 4096
```

The decompressor stub prepended to the final file must also fit within the 4096-byte limit. Total file = decompressor stub + compressed payload. A typical ZX0 decompressor is about 70 bytes in its smallest form, leaving roughly 4026 bytes for compressed data.

Competition Categories

Demoscene parties offer various size-limited categories beyond the classic 256. Common competition tiers include 4K, 8K, and sometimes 16K, alongside the smaller 256 and 512. The specific categories vary by party – Chaos Constructions, DiHalt, and Forever have all hosted 4K compos for the Spectrum. Some parties combine platforms (a "4K intro" compo accepting entries for any 8-bit platform), while others are Spectrum-specific. Check the party rules before starting – the measurement method (raw file size vs. loaded memory image) and the exact byte limit matter.

At 8K and 16K, the approach is essentially the same as 4K but with more breathing room. An 8K intro is a polished mini-demo where the compression pipeline is standard and the creative challenge is more about art direction than byte-counting. At 16K, you are essentially making a short demo that happens to fit in 16K – the size constraint shapes your ambition but does not dictate your instruction choices. The size-coding techniques from this chapter still help at these larger budgets, but their impact is proportionally smaller.

13.7 Practical: Writing a 256-Byte Intro Step by Step

Start with a working attribute plasma (~400 bytes) and optimise it to 256.
```

Step 1: The Unoptimised Version

A simple attribute plasma: fill 768 bytes of attribute memory with values from sine sums, offset by a frame counter. Sound: a cycling melody on AY channel A. This version is clean, readable, and roughly 400 bytes – the sine table (32 bytes), note table (16 bytes), inline AY writes, and the plasma loop with table lookups.

Step 2: Replace CALL with RST

Any call to a ROM address matching an RST vector saves 2 bytes per invocation. For AY output, replace the six verbose inline register writes (~60 bytes) with a small subroutine:

```
z80 id:ch13_step_2_replace_call_with_rst ay_write:                    ; register
in A, value in E     ld   bc, $FFFD     out  (c), a     ld   b, $BF     out
(c), e     ret                        ; 8 bytes total
```

Six calls (5 bytes each: load A + load E + CALL) = 30 + 8 = 38 bytes. Savings: ~22 bytes.

Step 3: Overlap Data with Code

The 32-byte sine table at the entry point decodes as mostly harmless Z80 instructions ($00=NOP, $06=LD B,n, $0C=INC C...). Place it at the program start. On first execution, the CPU stumbles through these "instructions," scrambling some registers. The main loop then jumps past the table and never executes it again – but the data remains for lookups. The table bytes serve double duty.

Step 4: Exploit Register State

After the plasma loop writes 768 attributes, HL = $5B00 and BC = 0 (from any LDIR used in initialisation). If the next operation needs these values, skip the explicit loads. Art-Top's discovery in NHBF was exactly this: register values from screen clearing matched the text string length. Not planned. Noticed.

After each optimisation pass, annotate what every register contains at every point. Register state is a shared resource – the fundamental currency of size-coding.

Step 5: Smaller Encodings Everywhere

- `LD A, 0` -> `XOR A` (save 1 byte)
- `LD HL, nn` + `LD A, (HL)` -> `LD A, (nn)` (save 1 byte if HL not needed)
- `JP` -> `JR` everywhere (save 1 byte each)
- `CALL sub : ... : RET` -> fall through directly (save 4 bytes)
- `PUSH AF` for temporary saves vs `LD (var), A` (save 2 bytes)

Step 6: Counting Bytes Precisely

Intuition about "how big is this" is unreliable. You need to count. There are three methods, and serious size-coders use all three.

Assembler output. sjasmplus can report the assembled size. The `DISPLAY` directive prints to the console during assembly, and `ASSERT` enforces the limit:

```
z80 id:ch13_step_6_counting_bytes intro_end:     ASSERT intro_end - init <= 256,
"Intro exceeds 256 bytes!"     DISPLAY "Intro size: ", /D, intro_end - init, "
bytes"
```

Run the assembler after every change. The DISPLAY line tells you where you stand; the ASSERT catches overflows before you waste time testing a broken binary.

Symbol file analysis. Assemble with `--sym=build/intro.sym` to get a symbol table. Compare label addresses to find exactly how many bytes each section occupies. When your intro is 262 bytes and you need to cut 6, the symbol file tells you that the AY init is 22 bytes (can you cut 2?), the effect loop is 38 bytes (can you merge the row and column counters?), the frame counter writeback is 8 bytes (can you restructure to make it 5?). Without this breakdown, you are guessing.

Hex dump inspection. After assembling, examine the raw binary in a hex editor (or `xxd build/intro.bin`). The hex dump shows you the actual bytes the CPU will execute. You will spot redundancies invisible in the source: two consecutive loads that could be one, an opcode whose value happens to match data you need elsewhere, a sequence of NOPs left by an accidental alignment. The hex dump is the ground truth. The source is an abstraction over it.

The Final Push

The last 10-20 bytes are the hardest. Structural rearrangement: reorder code so fall-throughs eliminate JR instructions. Merge the sound and visual loops. Embed data bytes in the instruction stream – if you need $07 as data and also need an `RLCA` (opcode $07), arrange for one to serve as both.

At this stage, keep a log. Write down every change you try: “moved AY init before pixel fill: saved 2 bytes (C register reuse), lost 1 byte (need extra LD B). Net: +1 byte.” Many changes do not help. Some make things worse. Without a log, you will try the same dead-end twice. With a log, you build a map of the solution space.

Try radical restructuring. Can the visual effect loop also update the AY? If the inner loop iterates 768 times (once per attribute cell), and you write a new tone value every 32 iterations (once per row), the sound update happens inside the visual loop at the cost of one `BIT 4, E` / `JR NZ` check – 4 bytes to merge two routines that previously needed separate framework code. Sometimes merging saves 10 bytes; sometimes it costs 5. You will not know until you try.

The escape hatch: choose a different effect. If your plasma needs a sine table and you are 30 bytes over, no amount of micro-optimisation will save you. Switch to an effect that generates its visual from pure register arithmetic: XOR patterns, modular arithmetic, bit manipulation. An XOR interference pattern like the one in `intro256.a80` needs zero data bytes. The visual is less smooth than a sine plasma, but it fits. At 256 bytes, “fits” is the only criterion that matters.

You stare at the hex dump. You try moving the sound routine before the visual routine. You try replacing the sine table with a runtime generator. Each attempt reshuffles the bytes. Sometimes everything lines up.

The satisfaction of fitting a coherent audiovisual experience into 256 bytes – of solving the puzzle – is real and specific and unlike any other feeling in programming.

13.8 Size-Coding Music: Bytebeat on AY

In the PC demoscene, **bytebeat** is a formula-driven approach to sound: a single expression like `t*((t>>12|t>>8)&63&t>>4)` generates PCM samples, producing surprisingly complex music from a few bytes of code. The concept was popularised by Viznut (Ville-Matias Heikkilä) in 2011, and 256-byte PC intros routinely use bytebeat for their soundtracks.

On the ZX Spectrum, the situation is different. The AY-3-8910 is not a DAC – it is a tone and noise generator with per-channel period and volume registers. You cannot feed it PCM samples in the traditional sense (volume-register sample playback exists but costs too many cycles for a size-coded intro). Instead, "AY bytebeat" means computing **tone periods and volume envelopes from mathematical formulas** driven by a frame counter.

The principle is the same as PC bytebeat: replace stored music data with a formula. The output target is different.

The Minimal AY Formula Engine

A typical approach in a 256-byte intro:

```
z80 id:ch13_the_minimal_ay_formula_engine ; Frame-driven AY "bytebeat" — ~20
bytes ; A = frame counter (incremented each HALT)     ld   e, a     and  $1F
; period = low 5 bits of frame     ld   d, a             ; D = tone period low
ld   a, e     rrca     rrca     rrca     and  $0F              ; volume = bits
5-7 of frame, shifted     ; Write to AY: register 0 = tone period low, register
8 = volume
```

This produces a cycling tone that sweeps through periods and fades in/out – not music in any traditional sense, but recognisably structured sound. The trick is choosing formulas that produce **musically interesting patterns** from simple bitwise operations.

Techniques for Better-Sounding Formulas

Pentatonic masking. Raw bitwise formulas produce chromatic noise. Mask the period value through a pentatonic lookup (5 bytes: the note intervals) to constrain output to a pleasant scale. Five bytes of data buys musically coherent sound.

Multi-channel formulas. The AY has three tone channels. Use different bit rotations of the same frame counter for each channel – they will produce related but distinct patterns, creating an impression of harmony:

```
z80 id:ch13_techniques_for_better     ld   a, (frame)     call .write_ch_a
; channel A: raw formula     ld   a, (frame)     rrca     rrca                  ;
channel B: frame >> 2     call .write_ch_b     ld   a, (frame)     add  a, a
; channel C: frame << 1     call .write_ch_c
```

Noise percussion. Toggle the noise generator on specific frame intervals (every 8th or 16th frame) for a rhythmic pulse. Cost: one `AND` + one `OUT` — about 6 bytes for a basic kick pattern.

LD A,R as entropy. The R register (memory refresh counter) is effectively random from a musical perspective. Mix it with the frame counter: `ld a,r : xor (frame)` pro-

duces evolving textures that never quite repeat. Useful for ambient or experimental soundscapes.

Bytebeat vs. Sequenced Music

	Bytebeat (formula)	Sequenced (pattern data)
Bytes	10-30 (code only)	200-400 (player) + 500+ (patterns)
Musical quality	Abstract, generative, alien	Melodic, structured, human
Best at	256b, 512b	1K, 4K
Sound	Rhythmic noise, sweeps, drones	Actual tunes

At 256 bytes, bytebeat is your only realistic option – there is no room for a pattern player. At 512, you can afford a tiny sequencer with 4-8 notes. At 4K, use a real player. The bytebeat approach is not inferior – it produces a *different kind* of sound that fits the aesthetic of tiny programs. Some of the most memorable 256-byte intros are memorable precisely because their sound is alien and generative, not because it imitates conventional music.

13.9 Procedural Graphics: The Rendered GFX Compo

Some demoscene parties run a **rendered graphics** (or **procedural graphics**) competition: submit a program that generates a static image. No pre-drawn bitmaps, no loaded data – every pixel must be computed. The visual output is judged as artwork, but it must be born from code.

On the Spectrum, this means your program fills the 6,912-byte screen area (bitmap + attributes) algorithmically, then halts. The image stays on screen for judging. File size limits vary – some compos allow any size, others impose 256-byte or 4K limits, turning it into a hybrid of size-coding and digital art.

Why the Spectrum Is Interesting for This

The Spectrum's display constraints – 1-bit pixels with 8×8 attribute colour – make procedural graphics a genuinely different challenge from doing it on a 256-colour VGA or 24-bit framebuffer. You cannot just compute RGB values per pixel. You must think in terms of:

- **Pixel patterns** within 8×8 character cells (dithering, halftone)
- **Attribute colour** per cell (2 colours from a palette of 15)
- **The interaction** between pixel pattern and attribute – a gradient needs both smooth dithering AND smooth attribute transitions

This constraint creates a distinctive visual style. Procedural Spectrum graphics look like nothing else – the colour grid gives them a mosaic quality that is part of the aesthetic, not a limitation to hide.

Common Approaches

Mandelbrot and Julia sets. The classic choice. The iteration loop is compact (~30-50 bytes for the core), and the fractal detail is infinite – zoom coordinates and iteration count are the only parameters. Map iteration count to dither pattern for pixel data, map to palette index for attributes. A Mandelbrot renderer fits comfortably in 256 bytes and produces images that look hand-crafted.

Interference patterns. Multiple overlapping sine or cosine waves, sampled at each pixel position. `pixel = sin(x*freq1 + phase1) + sin(y*freq2 + phase2) > threshold`. Produces organic, flowing shapes. On the Spectrum, threshold the sum to get the pixel bit, quantise to get the attribute colour.

Distance fields. Compute the distance from each pixel to a set of shapes (circles, lines, Bézier curves). Threshold the distance for pixel data, map it to colour for attributes. A few shapes can produce surprisingly complex images – overlapping circles alone can create intricate patterns.

L-systems and fractals. Recursive branching structures (trees, ferns, Sierpinski triangles). The recursion maps naturally to stack-based Z80 code, and the visual output has organic complexity from minimal code. A Sierpinski triangle renderer is about 20 bytes; a branching tree with random angles is perhaps 80.

The Byte Budget for Art

In a size-limited rendered GFX compo, every byte goes toward visual complexity. There is no frame loop, no sound, no animation – just a straight-line program that fills the screen and stops. This means your full budget goes to rendering code and coordinate generation. At 256 bytes, you can produce a detailed fractal. At 4K (compressed), you can generate images with multiple layers, computed textures, and careful dithering that approach hand-drawn quality.

The judging criterion is purely visual – the audience votes on the image, not the code. But the code constraint shapes the aesthetic. Procedural Spectrum graphics have a recognisable look: mathematical precision, fractal detail, and the characteristic colour grid of attribute-based rendering. The best entries embrace these constraints as style rather than fighting them.

13.10 Size-Coding as Art

Size-coding teaches you things that improve all your coding: the discipline of questioning every byte sharpens instruction-encoding awareness, the habit of looking for overlaps transfers to any optimisation work, and the practice of exploiting initial state and side effects makes you a better systems programmer.

Summary

- **Size-coding** competitions require complete programs in 256, 512, 1K, 4K, or 8K bytes – strict limits that demand a fundamentally different approach to

programming.

- **The size-coder's toolkit** includes register initialization assumptions, DJNZ as a combined decrement-and-branch, RST as a 1-byte CALL, overlapping instructions, and flag abuse via SBC A,A – tricks that save 1-5 bytes each but compound across a program.
- **NHBF** (UriS, CC 2025) demonstrates the 256-byte mindset: every byte does double duty, register states from one routine feed into the next, instruction choice is driven purely by encoding size.
- **The byte budget** for a typical 256-byte intro allocates ~90-130 bytes to framework (screen fill, AY init, frame sync, loop structure), leaving 120-160 bytes for the actual creative effect.
- **Choosing the right effect** matters more than micro-optimisation: attribute-based visuals with arithmetic formulas (XOR, modular math) encode cheaply; pixel-level effects and data tables consume too many bytes at 256.
- **The LPRINT trick** (diver4d, 2015) redirects BASIC's printer output to screen memory via address 23681, producing complex visual patterns in a handful of bytes – from pirated cassette loaders to demo art.
- **Each size tier is qualitatively different:** 256 bytes allows one effect with minimal sound; 512 adds sine tables and two-channel music; 1K enables pixel-level effects, tracker music, and multiple parts; 4K crosses the threshold into mini-demo territory with compression, full soundtracks, and multi-effect compositions.
- **4K intros** are where compression becomes viable: a ~200-byte decompressor unlocks 6-8K of working space, music players with pattern data fit comfortably, and scene tables enable 2-4 distinct effects with transitions. The optimisation target shifts from raw assembled size to compressed packed size.
- **AY bytebeat** replaces stored music data with formulas: compute tone periods and volumes from the frame counter using bitwise arithmetic. At 256 bytes, formula-driven sound (10-30 bytes) is the only option; at 4K, switch to a real pattern player. Pentatonic masking, multi-channel bit rotation, and noise percussion add musicality for minimal bytes.
- **Procedural graphics** (rendered GFX) competitions require every pixel to be computed, not loaded. The Spectrum's 1-bit pixels with 8×8 attribute colour make this a unique challenge – Mandelbrot sets, interference patterns, distance fields, and L-systems all produce distinctive results within the attribute grid aesthetic.
- **The optimisation process** moves from structural changes (eliminating tables, merging loops) to encoding choices (RST for CALL, JR for JP, XOR A for LD A,0) to serendipitous discoveries (register states aligning with data needs).
- **Counting bytes precisely** – via assembler DISPLAY/ASSERT, symbol file analysis, and hex dump inspection – is essential. Intuition about code size is unreliable.
- **The ORG trick** – choosing your load address so that address bytes double as useful data – represents the deepest level of the puzzle.

Try It Yourself

1. **Start large, shrink small.** Write an attribute plasma with a frame counter. Get it working at any size. Then optimise to 512 bytes, tracking every byte

saved and how.

2. **Explore LPRINT.** In BASIC, try `POKE 23681,64 : FOR i=1 TO 500 : LPRINT CHR$(RND*96+32); : NEXT i`. Watch the transposed data fill the screen. Experiment with different character ranges.
3. **Map your register state.** Write a small program and annotate what every register contains at every point. Look for places where one routine's output matches another's needed input.
4. **Study the RST vectors.** Disassemble the Spectrum ROM at $0000, $0008, $0010, $0018, $0020, $0028, $0030, $0038. These are your "free" subroutines.
5. **The 256-byte challenge.** Push the practical from this chapter to 256 bytes. You will need to make hard choices about what to keep and what to cut. That is the point.

Next: Chapter 14 – Compression: More Data in Less Space. We move from programs that fit in 256 bytes to the problem of fitting kilobytes of data into kilobytes of storage, with Introspec's comprehensive benchmark of 10 compressors as our guide.

Sources: UriS "NHBF Making-of" (Hype, 2025); diver4d "LPRINT Secrets" (Hype, 2015)

Chapter 14: Compression — More Data in Less Space

The ZX Spectrum 128K has 128 kilobytes of RAM. That sounds generous until you start subtracting: the screen takes 6,912 bytes (6,144 pixels + 768 attributes), the system variables claim their share, the AY music player and its pattern data want a bank or two, your code occupies another few thousand bytes, and the stack needs room to breathe. By the time you sit down to store the actual content of your demo — the graphics, the animation frames, the precalculated lookup tables — you are fighting for every byte.

A single full-screen image on the Spectrum is 6,912 bytes. A 4K intro can fit roughly 0.6 of one. A 48K demo could theoretically hold seven screens with nothing else. But demos are not slideshows. They have music. They have code. They have effects that demand tables of precalculated data. The question is not whether to compress — it is which compressor to use, and when.

This chapter is built around a benchmark. In 2017, Introspec (spke, Life on Mars) published "Data Compression for Modern Z80 Coding" on Hype — a meticulous comparison of ten compression tools tested against a carefully designed corpus. That article, with its 22,000 views and hundreds of comments, became the reference that ZX coders consult when choosing a compressor. We will walk through his results, understand the tradeoffs, and learn to pick the right tool for each job.

The Memory Problem

Let us be concrete about the constraints. Consider Break Space by Thesuper (Chaos Constructions 2016, 2nd place) — a demo with 19 scenes running on a ZX Spectrum 128K. One of those scenes, the Magen Fractal by psndcj, displays 122 frames of animation. Each frame is a full 6,912-byte screen. Uncompressed, that is 843,264 bytes — over six times the machine's total RAM.

psndcj compressed all 122 frames into 10,512 bytes. That is 1.25% of the original size. The entire animation, every frame of it, fits in less space than two uncompressed screens.

Another scene in Break Space, the Mondrian animation, packs 256 hand-drawn frames — every square cut separately, individually compressed — into 3 kilobytes.

These are not theoretical exercises. They are production techniques from a demo that competed at one of the scene's most prestigious parties. Compression is not

an optimisation you apply at the end. It is a fundamental architectural decision that determines what your demo can contain.

Compression as bandwidth amplifier

Introspec articulated the insight that elevates compression from a storage trick to a performance technique: **compression acts as a method to increase effective memory bandwidth**.

Suppose an effect needs 2 KB of data per frame. Store it compressed to 800 bytes and decompress with LZ4 at 34 T-states per output byte. The decompression costs 69,632 T-states — almost exactly one frame. But you can overlap it with border time, double-buffer a frame ahead, and interleave with effect rendering. The result: more data flowing through the system than the bus could deliver from uncompressed storage. The decompressor is a data amplifier.

The Benchmark

Introspec did not simply run each compressor on a few files and eyeball the results. He designed a corpus and measured systematically.

The corpus

The test data totalled 1,233,995 bytes across five categories:

- **Calgary corpus** — the standard academic compression benchmark (text, binary, mixed)
- **Canterbury corpus** — a more modern academic standard
- **30 ZX Spectrum graphics** — loading screens, multicolour images, game screens
- **24 music files** — PT3 patterns, AY register dumps, sample data
- **Miscellaneous ZX data** — tilemaps, lookup tables, mixed demo data

This mix matters. A compressor that excels on English text may struggle with ZX graphics, where long runs of zeros in the pixel area alternate with near-random attribute data. Testing on real Spectrum data — the data you will actually compress — is essential.

The results

Ten tools. Measured on total compressed size (lower is better), decompression speed in T-states per output byte (lower is faster), and decompressor code size in bytes (smaller is better for size-coded productions).

Tool	Compressed (bytes)	Ratio	Speed (T/byte)	Decompressor Size	Notes
Exomizer	596,161	48.3%	~250	~170 bytes	Best compression ratio
ApLib	606,833	49.2%	~105	199 bytes	Good balance
PuCrunch	616,855	50.0%	—	—	Complex LZ alternative
Hrust 1	613,602	49.7%	—	—	Relocatable stack depacker
Pletter 5	635,797	51.5%	~69	~120 bytes	Fast + decent compression
MegaLZ	636,910	51.6%	~130	~110 bytes	Revived by Introspec in 2019
ZX7	653,879	53.0%	~107	**69 bytes**	Tiny decompressor
ZX0	—	~52%	~100	**~70 bytes**	Successor to ZX7
LZ4	722,522	58.6%	**~34**	~100 bytes	Fastest decompression

Hrum	—	~52%	—	—	Declared obsolete

Only Exomizer broke the 600,000-byte barrier across the full corpus. But Exomizer's decompression speed — roughly 250 T-states per output byte — makes it impractical for anything that needs to decompress during playback.

The tradeoff triangle

Every compressor makes a tradeoff between three qualities:

1. **Compression ratio** — how small the compressed data gets
2. **Decompression speed** — how many T-states per output byte
3. **Decompressor code size** — how many bytes the decompression routine occupies

You cannot have all three. Exomizer wins on ratio but is slow to decompress and has a large decompressor. LZ4 is the fastest to decompress but wastes 10 percentage points of compression ratio. ZX7 has a 69-byte decompressor but compresses less aggressively than Exomizer.

Introspec's genius was to map these tradeoffs on a Pareto frontier — a curve where no tool can improve on one dimension without losing on another. If a compressor is dominated on all three axes by another tool, it is obsolete. If it sits on the frontier, it is the right choice for some use case.

"'mermaid id:ch14_the_tradeoff_triangle graph LR SRC["Source Data(raw bytes)"] –> EXO["Exomizer48.3% ratio~250 T/byte170B decompressor"] SRC –> APL["ApLib49.2% ratio~105 T/byte199B decompressor"] SRC –> PLT["Pletter 551.5% ratio~69 T/byte~120B decompressor"] SRC –> ZX0["ZX0~52% ratio~100 T/byte~70B decompressor"] SRC –> LZ4["LZ458.6% ratio~34 T/byte~100B decompressor"]

```
EXO --> T1["Best ratio<br>Slowest decompression"]
LZ4 --> T2["Worst ratio<br>Fastest decompression"]
ZX0 --> T3["Smallest decompressor<br>Good all-around"]

style EXO fill:#fdd,stroke:#333
style LZ4 fill:#ddf,stroke:#333
style ZX0 fill:#dfd,stroke:#333
style T1 fill:#fdd,stroke:#933
style T2 fill:#ddf,stroke:#339
style T3 fill:#dfd,stroke:#393


> **The tradeoff:** Smaller compressed size = slower decompression. No compressor wins on all
time loads, LZ4 for real-time streaming, ZX0 for size-coded intros.

His practical recommendations are crisp:

- **Maximum compression, speed irrelevant:** Exomizer. Use for one-time decompression at load
```

```
- **Good compression, moderate speed (~105 T/byte):** ApLib. A solid general-
purpose choice when you need decent ratio and can afford ~105 T-states per byte.
- **Fast decompression (~69 T/byte):** Pletter 5. When you need to decompress during gameplay
- **Fastest decompression (~34 T/byte):** LZ4. The only choice for real-time streaming --- dec
states per output byte, LZ4 can decompress over 2,000 bytes per frame. That is a 2 KB/frame da
- **Smallest decompressor (69--70 bytes):** ZX7 or ZX0. When the decompressor itself must be t
byte, 512-byte, or 1K intros where every byte of code counts.

Let these numbers guide your decisions. There is no universally "best" compressor. There is on

---

## How LZ Compression Works

All of the compressors in the table above belong to the Lempel-Ziv family. Understanding the c

LZ compression replaces repeated byte sequences with back-references. A match says: "copy N by
decoded stream." The compressed stream alternates between **literals** (raw bytes with no usef

The differences between compressors come down to encoding: how many bits for the offset, how m
length bit-level codes that compress tightly but require careful bit-extraction to decode ---
states per byte. LZ4 uses byte-aligned tokens that the Z80 processes with simple shifts and ma
states per byte at the cost of 10 percentage points of ratio. ZX0 uses single-
bit flags (0 = literal, 1 = match) with Elias interlaced codes for lengths, hitting a sweet sp

ZX Spectrum data compresses well because it has structure: large areas of identical bytes (bla
compressed data, and very short files where encoding overhead exceeds savings.

---

## ZX0 --- The Size Coder's Choice

ZX0, created by Einar Saukas, is the spiritual successor to ZX7 and has become the default com

### Why ZX0 exists

ZX7 was already remarkable: a 69-byte decompressor that achieved respectable compression ratio
byte range.

### The decompressor

The Z80 decompressor for ZX0 is hand-optimised assembly, designed specifically for the Z80's i

```z80 id:ch14_the_decompressor
; ZX0 decompressor — standard version
; HL = source (compressed data)
; DE = destination (output buffer)
; Uses: AF, BC, DE, HL
dzx0_standard:
        ld      bc, $ffff       ; initial offset = -1
        push    bc
```
```

```
        inc     bc              ; BC = 0 (literal length counter)
        ld      a, $80          ; bit buffer: only flag bit set
dzx0s_literals:
        call    dzx0s_elias     ; read literal length
        ldir                    ; copy literals
        add     a, a            ; read flag bit
        jr      c, dzx0s_new_offset
        call    dzx0s_elias     ; read match length
        ex      (sp), hl        ; retrieve offset from stack
        push    hl              ; put it back
        add     hl, de          ; calculate match address
        ldir                    ; copy match
        add     a, a            ; read flag bit
        jr      nc, dzx0s_literals
dzx0s_new_offset:
        ; ... offset decoding continues ...
```

Every instruction pulls double duty. The accumulator serves as both a bit buffer and a working register. The stack holds the last-used offset for repeated matches. The LDIR instruction handles both literal copying and match copying, keeping the code small.

At roughly 70 bytes, the entire decompressor fits in less space than a single ZX Spectrum character row. For a 256-byte intro, that leaves 186 bytes for everything else — the effect, the animation, the music. For a 4K intro, 70 bytes is negligible overhead. This is why ZX0 has become ubiquitous.

When to use ZX0

- **256-byte to 1K intros:** The tiny decompressor is essential. Every byte saved on the decompressor is a byte available for content.
- **4K intros:** ZX0 can decompress 4,096 bytes into 15–30 KB of code and data. SerzhSoft's Megademica (1st place, Revision 2019) used this exact strategy to fit what reviewers called "a full new-school demo" into a 4K intro.
- **General demo and game development:** When you need a solid all-around compressor with a small footprint. ZX0 is not the fastest decompressor, but it is fast enough for one-time decompression at load time, and its ratio is competitive with tools that have much larger decompressors.
- **RED REDUX** (2025) used the newer ZX2 variant (also by Saukas) to achieve the remarkable feat of including Protracker music in a 256-byte intro.

ZX0 is not the right choice for real-time streaming (use LZ4) or for maximum compression at any cost (use Exomizer). But for the vast majority of ZX Spectrum projects, it is the correct default.

RLE and Delta Coding

Not everything needs a full LZ compressor. Two simpler techniques handle specific data types more effectively.

RLE: Run-Length Encoding

The simplest scheme: replace a run of identical bytes with a count and a value. The decompressor is trivial:

```
z80 id:ch14_rle_run_length_encoding ; Minimal RLE decompressor — HL = source,
DE = destination rle_decompress:         ld      a, (hl)         ; read count
inc     hl         or      a         ret     z               ; count = 0 means
end         ld      b, a         ld      a, (hl)         ; read value         inc
hl .fill:  ld      (de), a         inc     de         djnz    .fill         jr
rle_decompress
```

Under 30 bytes of decompressor code. RLE compresses beautifully when data contains long runs — blank screens, solid-colour backgrounds, attribute fills. It compresses terribly on complex pixel art. The advantage over LZ: for size-coded intros where even ZX0's 70 bytes feel expensive, a custom RLE scheme frees precious space.

Delta coding: store what changed

Delta coding stores differences between consecutive values rather than absolute values. Two animation frames that are 90% identical? Store only the changed bytes — a list of (position, new_value) pairs. If only 691 bytes differ out of 6,912, the delta is 2,073 bytes (3 bytes per change) instead of a full frame. Apply LZ on top of the delta stream and it compresses further — the difference stream has more zeros and repeated small values than raw frame data.

The Break Space Magen Fractal exploits this: 122 frames at 6,912 bytes each, compressed to 10,512 bytes total, because each frame differs from the previous by a small amount. Delta + LZ is the standard pipeline for multi-frame animations, scrolling tilemaps, and sprite animations where the figure changes pose but the background stays fixed.

The Practical Pipeline

Understanding compression algorithms is useful. Integrating them into your build pipeline is essential.

From asset to binary

The pipeline: source asset (PNG) –> converter (png2scr) –> compressor (zx0) –> assembler (sjasmplus) –> .tap file. The compressor runs on your development machine, not the Spectrum. For ZX0: `zx0 screen.scr screen.zx0`. Include the result with sjasmplus's INCBIN directive:

```
z80 id:ch14_from_asset_to_binary compressed_screen:     incbin "assets/screen.zx0"
```

At runtime, decompress with a simple call:

```
z80 id:ch14_from_asset_to_binary_2     ld   hl, compressed_screen    ; source:
compressed data     ld   de, $4000                ; destination: screen memory
call dzx0_standard            ; decompress
```

Makefile integration

The compression step belongs in your Makefile, not in your head:

```
%.zx0: %.scr
    zx0 $< $@

demo.tap: main.asm assets/screen.zx0
    sjasmplus main.asm --raw=demo.bin
    bin2tap demo.bin demo.tap
```

Change a source PNG, run `make`, and the compressed binary is regenerated automatically. No manual steps, no forgotten recompression.

Example: loading screen with ZX0

A complete minimal example — decompress a loading screen to video memory and wait for a keypress:

“‘z80 id:ch14_example_loading_screen_with ; loading_screen.asm — assemble with sjasmplus org $8000 start: ld hl, compressed_screen ld de, $4000 call dzx0_standard

.wait: xor a in a, ($fe) cpl and $1f jr z, .wait ret

```
    include "dzx0_standard.asm"
```

compressed_screen: incbin “screen.zx0”

```
    display "Total: ", /d, $ - start, " bytes"


Use sjasmplus's DISPLAY directive to print size information during assembly. Always know exact

### Choosing the right compressor

Ask in order: (1) Size-coded intro? ZX0/ZX7 --- the 69--70 byte decompressor is non-
negotiable. (2) Real-time streaming? LZ4 --- nothing else is fast enough. (3) One-
time load? Exomizer --- maximum ratio, speed irrelevant. (4) Need a balance? ApLib or Pletter

---

## The MegaLZ Revival

In 2017, Introspec declared MegaLZ "morally obsolete." Two years later, he resurrected it hims

The insight: the compression *format* and the *decompressor implementation* are separable prob

- **Compact:** 92 bytes, ~98 T-states per byte
- **Fast:** 234 bytes, ~63 T-states per byte --- faster than three consecutive LDIRs

With these decompressors, MegaLZ "handily beats Pletter 5 and ZX7" on the combined ratio-
plus-speed metric. The lesson: do not assume a compressor is dead. The format is the hard part

---
```

What the Numbers Mean in Practice

4K intro: 4,096 bytes total. ZX0 decompressor: ~70 bytes. Engine + music + effects: ~2,400

Real-time streaming: You need 2 KB of data per frame at 50 fps. LZ4 at 34 T/byte decompres
states --- almost exactly one frame (69,888 T-states on 48K). Tight but feasible with overlapp
time decompression. ApLib would need 215,040 T-states for the same data --- over three frames.

128K multi-scene demo: Eight scenes, each with a 6,912-byte loading screen. Exomizer compr

256-byte intro: ZX0's 70-byte decompressor leaves 186 bytes for everything. More commonly
algorithmic data --- a colour ramp, a bitmap fragment --- ZX0 remains the tool.

Summary: The Compressor Cheat Sheet

Your situation	Use this	Why
One-time load, maximum ratio	Exomizer	48.3% ratio, speed irrelevant
General purpose, good balance	ApLib	49.2% ratio, ~105 T/byte
Need speed + decent ratio	Pletter 5	51.5% ratio, ~69 T/byte
Real-time streaming	LZ4	~34 T/byte, 2+ KB per frame
Size-coded intro (256b--1K)	ZX0 / ZX7	69--70 byte decompressor
4K intro	ZX0	Tiny decompressor + good ratio
Runs of identical bytes	RLE (custom)	Decompressor under 30 bytes
Sequential animation frames	Delta + LZ	Exploit inter-frame redundancy

The numbers are the answer. Not opinions, not folklore, not "I heard Exomizer is the best." In

---

Try It Yourself

1. **Compress a loading screen.** Take any ZX Spectrum .scr file (grab one from zxart.ee or cr
colour timing from Chapter 1.

2. **Measure the streaming limit.** Write a tight loop that decompresses data with the ZX0 sta

3. **Build a delta compressor.** Take two ZX Spectrum screens that differ slightly (save a gam

4. **Integrate compression into a Makefile.** Set up a project with a Makefile that automatica

> **Sources:** Introspec "Data Compression for Modern Z80 Coding" (Hype, 2017); Introspec "Com
saukas/ZX0)

\newpage

Chapter 15: Anatomy of Two Machines

```
> "Design characterizes realizational, stylistic, ideological integrity."
> -- Introspec (spke), "For Design" (Hype 2015)

Welcome to Part V. We are building a game.

Parts I through IV gave you the demoscener's toolkit: cycle counting, screen tricks, inner loo

This chapter is the hardware reference for everything that follows. Where Chapter 1 gave you t

---

## 15.1 ZX Spectrum 128K: Memory Map

The original 48K Spectrum had a simple memory model: 16 KB ROM at `$0000`-
`$3FFF`, 48 KB RAM at `$4000`-`$FFFF`. The 128K model keeps this layout visible to the CPU but

The 128K has eight 16 KB pages of RAM (pages 0-7, totalling 128 KB) and two 16 KB ROMs (ROM 0:

| Address Range | Contents | Notes |
|---------------|----------|-------|
| `$0000`-`$3FFF` | ROM (0 or 1) | Selected by bit 4 of `$7FFD` |
| `$4000`-`$7FFF` | RAM page 5 | **Always** page 5. Screen memory lives here. |
| `$8000`-`$BFFF` | RAM page 2 | **Always** page 2. |
| `$C000`-`$FFFF` | RAM page N | Switchable: any page 0-7 via `$7FFD` |

Pages 5 and 2 are hardwired into their slots. You cannot swap them out. This means the screen
`$57FF`) is always accessible, and your main code (typically ORG'd at `$8000`) sits in page 2

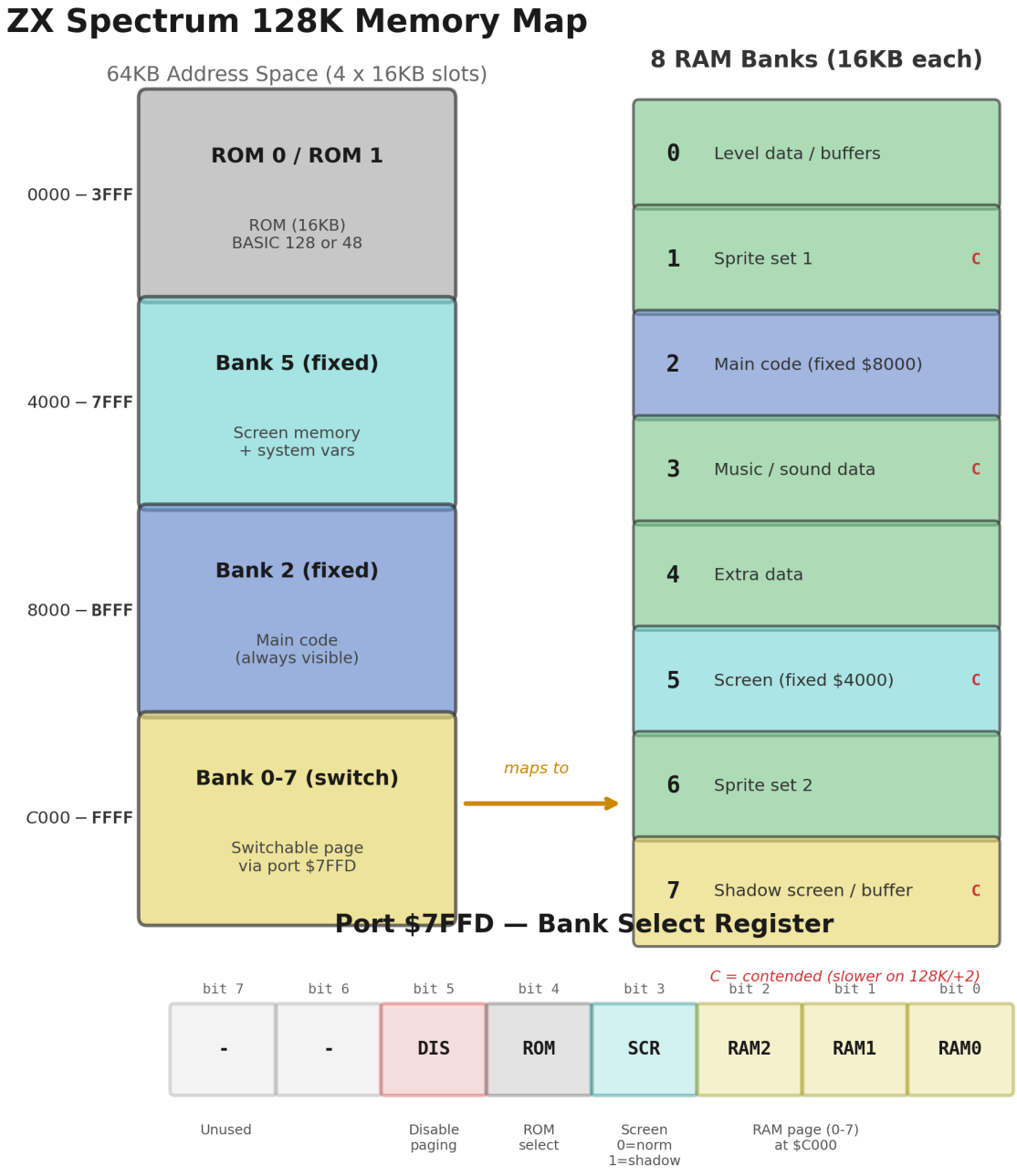
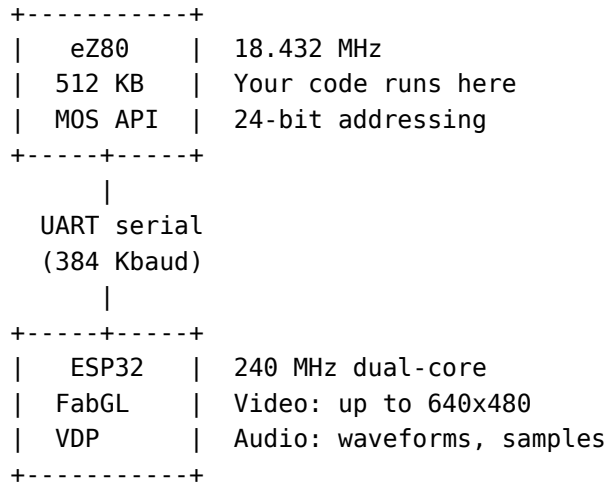
The top 16 KB slot at `$C000`-`$FFFF` is the flexible one. Write to port `$7FFD` and the page

### The $7FFD Port: Bank Switching

Port `$7FFD` controls memory configuration on the 128K. It is write-only -- you cannot read it

```text id:ch15_the_7ffd_port_bank_switching
Port $7FFD bit layout:
  Bit 0-2:  RAM page mapped at $C000 (0-7)
  Bit 3:    Screen select (0 = normal screen at page 5,
                           1 = shadow screen at page 7)
  Bit 4:    ROM select (0 = 128K ROM, 1 = 48K BASIC ROM)
  Bit 5:    Disable paging (set this and banking is locked
            until next reset -- used by 48K BASIC)
  Bits 6-7: Unused
```

A typical bank-switch routine:

```z80 id:ch15_the_7ffd_port_bank_switching_2
; Switch RAM page at $C000 to page number in A (0-7)
; Preserves other $7FFD bits from shadow variable bank_switch:
ld   b, a                ; 4T   save desired page
ld   a, (bank_shadow)    ; 13T  load current $7FFD state
and  %11111000           ; 7T   clear page bits (0-2)
or   b                   ; 4T   insert new page number
ld   (bank_shadow), a    ; 13T  update shadow
ld   bc, $7FFD           ; 10T
```
```

```
out  (c), a              ; 12T  do the switch     ret                       ;
10T                               ; --- 73T total
```

Those 73 T-states are not free, but they are negligible compared to a frame budget of 70,000+. The real cost of banking is architectural, not temporal: you must design your data layout so that you never need to access two different banked pages simultaneously. Music data in page 4, level data in page 3, sprite graphics in page 6 – but your music player and your renderer cannot both be running from `$C000` at the same time.

The shadow screen. Bit 3 of `$7FFD` selects which RAM page the ULA reads for display: page 5 (normal) or page 7 (shadow). This gives you hardware double-buffering – draw into the shadow screen while the ULA displays the normal one, then flip by toggling bit 3. We will use this heavily in Chapter 17 (Scrolling) and Chapter 21 (Full Game).

A Practical Memory Map for a 128K Game

Here is how a real game might lay out its 128 KB across the eight pages:

Page	Slot	Usage
0	`$C000` (banked)	Level data (maps, tile definitions)
1	`$C000` (banked)	Sprite graphics set 1
2	`$8000-$BFFF` (fixed)	Main game code, entity system, interrupt handler
3	`$C000` (banked)	Sprite graphics set 2, lookup tables
4	`$C000` (banked)	Music data (.pt3 patterns, instruments)
5	`$4000-$7FFF` (fixed)	Primary screen, attribute memory, system vars
6	`$C000` (banked)	Sound effects, additional level data
7	`$C000` (banked)	Shadow screen (double-buffer target)

Notice: page 7 serves double duty. The ULA can display it as the shadow screen, but you can also bank it into `$C000` and use it as a 16 KB data page when you are not double-buffering. Many demos exploit this.

The critical constraint: **your interrupt handler and main loop must live in pages 2 or 5**, because those are the only pages guaranteed to be mapped at all times. If an interrupt fires while page 4 is banked in at `$C000`, and your interrupt handler lives at `$C000`, the CPU jumps into your music data instead of your code. The result is a crash, usually a spectacular one.

Rule: never put time-critical code in a banked page unless you are absolutely certain which page is active when that code runs.



Figure 10: ZX Spectrum 128K memory map

15.2 Contended Memory: The Practical Truth

In Chapter 1, we established that Pentagon clones have no contended memory and that cycle counts are reliable everywhere. That is true, and the Pentagon remains the standard for cycle-counted demo work. But if you are writing a game for release, your players will include people running original Sinclair hardware, Amstrad +2A/+3 models, and modern FPGA clones that emulate the original timing. You need to know what contended memory does and how to avoid its worst effects.

Introspec covered this exhaustively in his "GO WEST" articles on Hype (2015). Here is the practical summary.

What Gets Contended

On the original Sinclair machines, the ULA and the CPU share a memory bus. When the ULA is reading screen data to paint the display (during the 192 active scanlines), any CPU access to certain RAM pages gets delayed. The CPU is literally halted for extra T-states while the ULA finishes its read.

The contended pages differ between models:

Model	Contended Pages	Always Fast
48K	Page 5 only (`$4000-$7FFF`)	`$8000-$FFFF` (uncontended)
128K / +2	Pages 1, 3, 5, 7	Pages 0, 2, 4, 6
+2A / +2B / +3	Pages 4, 5, 6, 7	Pages 0, 1, 2, 3

The 48K contends only the lower 16 KB of RAM (`$4000-$7FFF`, page 5) – the upper 32 KB (`$8000-$FFFF`) is uncontended. On the 128K, the pattern is every odd page. On the +2A/+3, it flips: the high pages are contended.

This has immediate practical consequences. On a 128K, your main code at `$8000` (page 2) is in uncontended memory – fast. The screen at `$4000` (page 5) is contended – writes to screen memory are slower during active display. And page 7 (the shadow screen) is also contended, which means double-buffer fills to the shadow screen are slower than you might expect on original hardware.

How Much Slower?

Introspec measured the actual penalties:

- **Random byte access to contended memory:** approximately **0.92 extra T-states per byte** on average during active display
- **Stack operations (PUSH/POP) to contended memory:** approximately **1.3 extra T-states per byte** on average
- **During border time: zero penalty** – contention only occurs while the ULA is actively painting scanlines

That 0.92 figure means a `LD A,(HL)` that should cost 7 T-states will cost, on average, about 7.92 T-states when HL points into contended memory during active display. A PUSH that writes two bytes to contended memory at 11 T-states will cost about 13.6 T-states instead.

These averages hide a messy reality: the actual penalty depends on where in the ULA's 8-T-state read cycle your CPU access lands. The pattern repeats every 8 T-states: penalties of 6, 5, 4, 3, 2, 1, 0, 0 extra states. You can land anywhere in this cycle, and the penalty compounds with each memory access within an instruction. This makes precise cycle counting on contended machines genuinely difficult.

The Practical Response

For game development, not demo effects, the approach is straightforward:

1. **Put your code in uncontended memory.** On the 128K, ORG at `$8000` (page 2) – always fast.
2. **Write to the screen during border time when possible.** The top and bottom borders give you contention-free access to screen memory. So does the left/right border of each scanline.
3. **Do not worry about precise contention modelling.** Budget a 15-20% slowdown for code that touches screen memory during active display, and design your frame budget with that margin. This is not cycle-counted demo work; this is game development.
4. **Test on real hardware or accurate emulators.** Fuse emulates contention correctly. Unreal Speccy (Pentagon mode) does not, by design. ZEsarUX can emulate multiple models.

Introspec's advice from GO WEST boils down to this: **contended memory is a portability issue, not a drama.** If your code runs on Pentagon, it will almost certainly run on original hardware too – just a little slower during screen writes. The places where contention actually breaks things are cycle-exact raster effects (multicolour, floating bus sync), and those are demo techniques, not game techniques.

15.3 ULA Timing

The ULA generates the video signal and the CPU interrupt. Understanding its timing is essential for border effects, interrupt-driven music, and screen synchronisation.

Frame Structure

A complete frame consists of scanlines. The scanline width and total scanline count differ between models:

Machine	T-states/line	Scanlines	T-states/frame
ZX Spectrum 48K	224	312	69,888
ZX Spectrum 128K	228	311	70,908
Pentagon 128	224	320	71,680

Note the 128K's wider scanline (228 vs 224 T-states). The extra 4 T-states per line are in the border/sync portion, not the active display.

Tact-Maps: Frame Regions

The frame divides into three regions. The interrupt fires at the start of vertical blank, before the top border. Here is the timing map for each model:

Pentagon 128 (71,680 T-states)

```
Interrupt ──┐
            │
Top border  │  80 lines × 224T = 17,920T   No screen reads. No contention.
            │
Active      │ 192 lines × 224T = 43,008T   ULA reads screen memory.
display     │                                No contention on Pentagon.
            │
Bottom      │  48 lines × 224T = 10,752T   No screen reads. No contention.
border      │
────────────┘  Total: 71,680T
```

ZX Spectrum 128K (70,908 T-states)

```
Interrupt ──┐
            │
Top border  │  63 lines × 228T = 14,364T   No screen reads. No contention.
            │
Active      │ 192 lines × 228T = 43,776T   ULA reads screen memory.
display     │                                Contention on pages 1,3,5,7.
            │
Bottom      │  56 lines × 228T = 12,768T   No screen reads. No contention.
border      │
────────────┘  Total: 70,908T
```

ZX Spectrum 48K (69,888 T-states)

```
Interrupt ──┐
            │
Top border  │  64 lines × 224T = 14,336T   No screen reads. No contention.
            │
Active      │ 192 lines × 224T = 43,008T   ULA reads screen memory.
display     │                                Contention on all RAM.
            │
Bottom      │  56 lines × 224T = 12,544T   No screen reads. No contention.
border      │
────────────┘  Total: 69,888T
```

After a `HALT`, you have the entire top border period – 17,920 T-states on Pentagon, 14,364 on 128K – to do work before the beam enters the active display area and contention begins. This is why well-structured Spectrum code does screen writes at the top of the frame: you get contention-free access to screen memory during the border period.

Scanline Timing

Each scanline breaks down into an active portion (where the ULA reads screen data) and border/sync portions:

48K and Pentagon (224 T-states per line):

```
128T  active pixel area (ULA reads screen data)
 24T  right border
 48T  horizontal sync + retrace
 24T  left border
```

128K (228 T-states per line):

```
128T  active pixel area (ULA reads screen data)
 24T  right border
 52T  horizontal sync + retrace
 24T  left border
```

During the 128 active T-states, memory access to contended pages gets delayed (on non-Pentagon machines). During the remaining 96 T-states (or 100 on 128K), no contention. Even during active display, roughly half of each scanline is contention-free.

Total vs Practical Budget

The frame totals above are the time between interrupts. The *practical* budget – T-states available for your code – is less:

Overhead	Cost
HALT + interrupt acknowledge (IM1)	~30 T-states
Minimal ISR (EI + RET)	~14 T-states
Typical PT3 music player (in ISR)	~3,000–5,000 T-states
Main loop housekeeping (frame counter, HALT jump)	~20–50 T-states

Practical budgets with a music player running:

Machine	Total	After PT3 player	After player + contention margin
Pentagon	71,680	~66,000–68,000	~66,000–68,000 (no contention)
128K	70,908	~65,000–67,000	~55,000–60,000 (screen writes during active display)
48K	69,888	~64,000–66,000	~50,000–55,000 (all RAM contended)

When this book says “frame budget of ~70,000 T-states,” it means the total. When planning your inner loops, budget for the practical figure – typically 65,000–68,000 on Pentagon with music.

15.4 Floating Bus, ULA Snow, and the $7FFD Bug

These are three hardware quirks that appear on original Sinclair machines but not on most clones. You may never encounter them in game development, but they can cause mysterious bugs if you do not know they exist.

Floating Bus

On original Spectrum hardware, reading from an unattached port returns whatever data the ULA happens to be putting on the data bus at that moment. During active display, the ULA is reading screen memory, so a read from port `$FF` returns the byte the ULA is currently reading.

Demo coders exploit this for beam synchronisation: read the floating bus in a tight loop until you see a known value from screen memory, and you know exactly where the beam is. This is the cheapest sync method – no interrupt timing required.

Games rarely need this, but be aware: if your code reads from a port that does not exist on the hardware, the return value is unpredictable and varies between models. The floating bus is *not* emulated on Pentagon, Scorpion, or ZX Next.

The $7FFD Read Bug

Port `$7FFD` is write-only. But on some Spectrum models, reading from port `$7FFD` (even unintentionally, through an instruction that happens to put `$7FFD` on the address bus) causes the floating bus value to be written into the port. This triggers a spurious page switch.

The practical danger: the Z80 instruction `LD A,(nn)` puts the address `nn` on the bus during execution. If `nn` happens to be `$7FFD` and you are reading data stored at address `$7FFD`, the memory read can trigger a port write on original hardware. This is an obscure bug but a real one. Avoid storing data at address `$7FFD`.

ULA Snow

If the Z80's I register (used for IM2 interrupt vector table base) is set to a value in the range `$40`-`$7F`, the DRAM refresh cycle during every M1 opcode fetch puts an address in the `$4000`-`$7FFF` range onto the address bus. This conflicts with the ULA's screen reads and produces visual "snow" – random noise on the display.

The fix is simple: **never set I to a value between `$40` and `$7F`.** The typical IM2 setup uses `I = $FE` with a 257-byte table of identical vectors at `$FE00`-`$FF00`. This keeps I well above the danger zone.

15.5 Clone Differences

The ZX Spectrum ecosystem includes dozens of clones, but three matter most for modern development: the Pentagon 128, the Scorpion ZS-256, and the ZX Spectrum Next.

Pentagon 128

The Pentagon is the standard platform for the Russian demoscene and the primary target for this book's demoscene chapters.

Parameter	Pentagon 128	Original 128K
CPU clock	3.5 MHz	3.5 MHz
T-states per frame	**71,680**	70,908
Scanlines per frame	**320**	311
Contended memory	**None**	Pages 1, 3, 5, 7
Border lines (top)	**80**	63
Border lines (bottom)	**48**	56

The extra 772 T-states per frame (71,680 vs 70,908) come from the additional scanlines. The border is distributed differently: a taller top border and a shorter bottom border. This affects border effects – demo code that produces a symmetrical border pattern on the 128K will be slightly asymmetric on the Pentagon.

The absence of contended memory is the Pentagon's defining feature for programmers. Every instruction costs exactly what the datasheet says. This is why we use Pentagon timing throughout this book.

7 MHz Turbo mode. Many Pentagon-compatible machines (Pentagon 512, Pentagon 1024, ATM Turbo 2+) offer a 7 MHz turbo mode. The CPU runs at double speed, but the ULA timing stays the same. This means the frame budget doubles to approximately 143,360 T-states in turbo mode. The catch: turbo mode is not standard across all machines, and code that relies on it will not run on a stock Pentagon 128 or any Sinclair hardware.

For games, turbo mode is a luxury that lets you run more complex logic or more sprites per frame. For demos targeting compo rules, it is usually forbidden – competitions specify "Pentagon 128K, 3.5 MHz."

Scorpion ZS-256

The Scorpion is a Ukrainian clone with 256 KB of RAM (16 pages of 16 KB) and several hardware extensions.

Feature	Scorpion ZS-256
RAM	256 KB (16 pages)
Banking	Extended `$1FFD` port for pages 8-15
Graphics	GMX mode: 320x200, 16 colours from 256
Contended memory	None
Frame timing	Pentagon-compatible (71,680 T-states)

The doubled RAM is useful for games: you get 16 data pages instead of 8. The extra pages are accessed via port `$1FFD`, which uses a similar scheme to `$7FFD` but controls the additional RAM.

GMX (Graphics Mode Extended) is the Scorpion's party trick: a 320x200 display with 16 colours chosen from a 256-colour palette. This breaks completely with the Spectrum's attribute-based display, offering a linear framebuffer closer to what you might see on an Amiga or a PC VGA. The GMX framebuffer is large (32,000 bytes for 4-bit colour) and lives in the extended RAM pages.

Few games target GMX because it limits your audience to Scorpion owners. But it demonstrates what Z80 hardware can do when freed from the ULA's attribute grid.

ZX Spectrum Next

The ZX Spectrum Next is the modern flagship of the platform: an FPGA-based machine that is backward-compatible with the original Spectrum but adds substantial new hardware.

Feature	ZX Spectrum Next
CPU	Z80N (Z80 + new instructions) at 3.5 / 7 / 14 / 28 MHz
RAM	1 MB (extendable to 2 MB), 8 KB MMU pages
MMU	8 slots x 8 KB = fine-grained memory mapping
Layer 2	256x192 or 320x256, 8-bit colour (256 colours)
Tilemap	Hardware tilemap layer, 40x32 or 80x32 tiles
Sprites	128 hardware sprites, 16x16, up to 12 per scanline
Copper	Co-processor for per-scanline register changes
DMA	zxnDMA for fast block transfers
AY sound	3 x AY-3-8910 (9 channels) with per-channel stereo panning

The Next's **MMU** is fundamentally different from the 128K's banking. Instead of one switchable 16 KB slot, the Next divides the entire 64 KB address space into eight 8 KB slots. Each slot can be independently mapped to any 8 KB page from the 1-2 MB RAM pool. This means you can have fine-grained control:

```
z80 id:ch15_zx_spectrum_next ; Map 8KB page $0A into slot 3 ($6000-$7FFF)     ld
a, $0A     ld   bc, $243B          ; Next register select port     ld   a, $53
; Register $53 = MMU slot 3     out  (c), a     ld   bc, $253B          ; Next
register data port     ld   a, $0A             ; Page $0A     out  (c), a
```

This is much more flexible than the 128K's single switchable slot. You can map sprite data into one 8 KB window, level data into another, and music data into a third – all simultaneously visible.

Layer 2 gives you a 256-colour bitmap display without attribute clash. This is the single biggest quality-of-life improvement for game developers: no more careful attribute planning, no more colour clash workarounds. Just a framebuffer where each byte is one pixel. The cost is memory: a 256x192 Layer 2 screen is 49,152 bytes.

Hardware sprites on the Next provide 128 sprite slots, each 16x16 pixels with 8-bit colour, up to 12 per scanline. Sprite attributes (position, pattern, rotation) are set through Next registers and port `$57`. No software rendering needed.

The Copper is a co-processor that executes a simple program synchronised to the beam position. It can write to any Next register at any scanline, enabling per-line palette changes, scroll offsets, and raster effects without consuming Z80 T-states – a deliberate homage to the Amiga Copper.

zxnDMA provides hardware-accelerated block transfers at approximately 2 T-states per byte – about 10 times faster than `LDIR`. For filling the Layer 2 framebuffer or transferring sprite data, DMA is transformative.

The Next is essentially a different machine that happens to be backward-compatible. The interesting constraints shift from "can I fit this in the frame budget" to "how

do I best use multiple hardware layers.”

15.6 Agon Light 2: A Different Beast

The Agon Light 2 is the second platform for our game development chapters. It runs a Zilog eZ80 – a direct descendant of the Z80 – at 18.432 MHz, with 512 KB of flat RAM and a separate ESP32 co-processor handling video and audio. The architecture is fundamentally different from the Spectrum: instead of a CPU that shares a bus with a fixed video output chip, the Agon uses two independent processors communicating over a serial link.

Dual-Processor Architecture

The Agon’s defining characteristic is the split between the **eZ80** (your CPU) and the **ESP32** (the VDP, Video Display Processor):



This split has important consequences:

1. **No shared video memory.** You cannot write directly to a framebuffer. Every pixel, every sprite, every tile operation is a *command* sent over the serial link from the eZ80 to the ESP32.
2. **Latency.** The serial link runs at 384,000 baud. A single command byte takes about 26 microseconds to transmit. Complex drawing operations (fill rectangle, draw bitmap) require multiple bytes and the VDP needs time to execute them.
3. **Asynchronous rendering.** The VDP processes commands from a buffer. Your eZ80 code sends commands and continues running. The VDP catches up independently. This means you do not have the Spectrum’s tight coupling between CPU work and screen output – but you also cannot precisely control when pixels appear.
4. **Independent frame rate.** The VDP renders at its own rate (typically 60 Hz). Your eZ80 game loop can run at whatever rate it wants; the VDP will display whatever it has most recently drawn.

For Spectrum programmers, this is a different approach entirely. You go from “I write bytes to video memory and they appear on the next scanline” to “I send

drawing commands and trust the VDP to render them eventually." The upside is enormously reduced CPU overhead for graphics. The downside is less control.

eZ80 Memory Model: 24-Bit Flat

The eZ80 has a 24-bit address bus, giving it a theoretical 16 MB address space. The Agon Light 2 maps 512 KB of RAM into the bottom of this space:

Address Range	Size	Contents
`$000000-$07FFFF`	512 KB	RAM
`$080000-$0FFFFF`	512 KB	RAM (mirror, on some boards)
`$A00000-$FFFFFF`	varies	I/O, on-chip peripherals

No banking. No page switching. No contended memory. Your code, your data, your buffers, your lookup tables – everything lives in one flat, linearly addressable space. After the Spectrum's 8-page juggling act, the simplification is immediate.

The eZ80 supports two operating modes that determine how it uses this address space.

ADL Mode vs Z80 Mode

This is the most important architectural distinction on the Agon, and it trips up newcomers constantly.

Z80 mode (also called Z80-compatible mode) makes the eZ80 behave like a classic Z80: 16-bit registers, 16-bit addresses, 64 KB address space. All standard Z80 code runs unmodified. The upper 8 bits of the address come from the MBASE register, creating a 64 KB "window" into the 24-bit address space. This is what you use when porting existing Z80 code.

ADL mode (Address Data Long) is the eZ80's native mode: 24-bit registers, 24-bit addresses, full 16 MB address space. HL, BC, DE, SP, and IX/IY are all 24 bits wide. `LD HL,$123456` loads a 3-byte value. `PUSH HL` pushes 3 bytes onto the stack (not 2). Every pointer is 3 bytes.

```
z80 id:ch15_adl_mode_vs_z80_mode ; ADL mode: 24-bit addressing, full 512KB
accessible     ld   hl, $040000       ; point to a buffer 256KB into RAM     ld
(hl), $FF         ; write directly -- no banking needed     ld   bc, 1024     ld
de, $040001     ldir                    ; fill 1KB in one shot
```

MOS (the Agon's operating system) boots the eZ80 in ADL mode, and most Agon software stays in ADL mode. The key differences from Z80 mode:

Feature	Z80 Mode	ADL Mode
Register width	16 bits	24 bits
Address space	64 KB (via MBASE)	16 MB (24-bit)
PUSH/POP size	2 bytes	3 bytes
JP/CALL addresses	16-bit	24-bit
Stack frame size	2 bytes per entry	3 bytes per entry
Instruction encoding	Z80-compatible	Extended (3-byte addresses)

The trap: if you write code assuming 16-bit values and run it in ADL mode, things break in subtle ways. A `PUSH HL` pushes 3 bytes, not 2, so your stack-based data structures have a different size. A `JP (HL)` jumps to a 24-bit address, so lookup tables of 16-bit addresses will not work. The eZ80 provides `LD.S` and `LD.L` suffixed instructions to explicitly control the data width, and you can switch between modes with `JP.LIL` / `JP.SIS` prefixes, but this gets complicated fast.

The practical rule for games: stay in ADL mode. Use 24-bit addresses everywhere. Do not try to share code between a Spectrum build and an Agon build at the source level – the addressing is too different. Instead, share *algorithms* and *data formats*, with platform-specific implementations for memory access, I/O, and graphics.

MOS API: The Operating System

MOS (Machine Operating System) provides system services on the Agon: file I/O, keyboard input, timer access, and VDP communication. MOS calls are made through `RST $08` with a function number in the A register:

```
z80 id:ch15_mos_api_the_operating_system ; MOS API: open a file     ld   hl,
filename       ; pointer to null-terminated filename     ld   c, $01            ;
mode: read     rst  $08                ; MOS call     db   $0A               ;
function $0A: ffs_fopen     ; Returns file handle in A filename:     db   "level1.dat",
0
```

Key MOS functions for game development:

Function	Code	Description
`mos_getkey`	`$00`	Read keyboard (non-blocking)
`mos_load`	`$01`	Load file from SD card
`mos_save`	`$02`	Save file to SD card
`mos_sysvars`	`$08`	Get pointer to system variables (vsync counter, etc.)
`ffs_fopen`	`$0A`	Open file
`ffs_fclose`	`$0B`	Close file
`ffs_fread`	`$0C`	Read from file
`mos_getrtc`	`$12`	Get real-time clock

File I/O on the Agon is trivially easy compared to the Spectrum. No tape loading, no esxDOS wrappers, no TR-DOS: just open a file from the SD card and read it into memory. Level data, sprite sheets, music – load them on demand, no banking gymnastics.

VDP Commands: Talking to the Screen

All graphics go through VDU commands sent to the ESP32 VDP. The eZ80 sends bytes to a VDU output stream; the VDP interprets them as drawing instructions:

```
z80 id:ch15_vdp_commands_talking_to_the ; VDP: draw a filled rectangle at (10,
10)     rst  $10 : db 25        ; PLOT command     rst  $10 : db 85        ;
mode: filled rectangle     rst  $10 : db 10        ; x low     rst  $10 : db 0
```

```
; x high     rst  $10 : db 10        ; y low     rst  $10 : db 0         ; y
high
```

Verbose compared to `LD (HL),A`, but the VDP does the rendering on the ESP32. The VDP supports bitmap modes (up to 640x480), up to 256 hardware sprites (each up to 64x64), hardware tilemaps with scrolling, and audio (waveforms, ADSR, samples).

The bottleneck is the serial link, not the CPU. A complex scene with many sprite updates can saturate the UART, causing visual lag. Minimise VDP commands per frame: batch updates, use hardware scrolling instead of redrawing tiles, and let the sprite engine do the heavy lifting.

15.7 Comparing the Platforms

Let us lay out the two machines side by side, focusing on what matters for the game engine we will build in Chapters 16-19.

Feature	ZX Spectrum 128K	Agon Light 2
CPU	Z80A @ 3.5 MHz	eZ80 @ 18.432 MHz
T-states per frame	~70,908 (128K, 50 Hz) / 71,680 (Pentagon, 50 Hz)	~307,200 (60 Hz)
RAM	128 KB (8 x 16 KB pages)	512 KB (flat)
Address space	64 KB (banked)	16 MB (24-bit)
Screen memory	Shared bus, direct write	Separate VDP, command-based
Colours	15 (8 base x bright, minus overlap)	Up to 64 in standard modes
Resolution	256x192 (attribute colour per 8x8)	Configurable, up to 640x480
Sprites	Software only	Up to 256 hardware sprites
Scrolling	Software only (manual shift/copy)	Hardware scroll offsets
Sound	AY-3-8910 (3 channels)	ESP32 audio (multi-channel, waveforms)
Storage	Tape / DivMMC (esxDOS)	SD card (FAT32)
Double buffering	Shadow screen (page 7)	VDP-managed

The frame budget ratio is approximately 4:1 in the Agon's favour. But the Agon's graphics go through a serial bottleneck, so raw CPU speed does not translate directly to rendering speed. On the Spectrum, `PUSH HL` writes two bytes to the screen in 11 T-states. On the Agon, updating a sprite position requires 6+ bytes over a 384 Kbaud link, taking hundreds of microseconds regardless of CPU speed.

The Spectrum rewards byte-level optimisation. The Agon rewards architectural decisions. Both reward careful thinking about frame budgets.

15.8 Practical: Memory Inspector Utility

Let us build a simple memory inspector for both platforms. This utility displays a region of RAM as hex bytes on screen, and lets you navigate through memory with the keyboard. It is the kind of tool you will use constantly during development.

Spectrum Version

The Spectrum version writes directly to screen memory. We display 16 rows of 16 bytes (256 bytes per page) with the start address shown at the left.

“‘z80 id:ch15_spectrum_version ; Memory Inspector - ZX Spectrum 128K ; Displays 256 bytes of memory as hex, navigable with keys ; ORG $8000 (page 2, uncontended)

```
ORG  $8000
```

SCREEN_ATTR EQU $5800 START_ADDR EQU inspect_addr ; address to inspect (self-mod)

start: call clear_screen

main_loop: halt ; sync to frame

```
; Read keyboard
call read_keys                ; returns: A = action
cp   1
jr   z, .page_up             ; Q = previous page
cp   2
jr   z, .page_down           ; A = next page
cp   3
jr   z, .bank_up             ; P = next bank
cp   4
jr   z, .bank_down           ; O = previous bank
jr   .draw
```

.page_up: ld hl, (inspect_addr) ld de, -256 add hl, de ld (inspect_addr), hl jr .draw .page_down: ld hl, (inspect_addr) ld de, 256 add hl, de ld (inspect_addr), hl jr .draw .bank_up: ld a, (current_bank) inc a and 7 ; wrap 0-7 ld (current_bank), a call bank_switch jr .draw .bank_down: ld a, (current_bank) dec a and 7 ld (current_bank), a call bank_switch

.draw: ; Display current bank and address call draw_header

```
; Display 16 rows x 16 bytes
ld   hl, (inspect_addr)
ld   b, 16                    ; 16 rows
ld   de, $4060                ; screen position (row 3, col 0)
```

.row_loop: push bc push hl

```
; Print address
ld   a, h
call print_hex                ; print high byte of address
ld   a, l
call print_hex                ; print low byte
ld   a, ':'
call print_char
```

```
; Print 16 hex bytes
pop  hl
push hl
ld   b, 16
```

.byte_loop: ld a, (hl) call print_hex ; 17T call + print routine inc hl ld a, ' ' call print_char djnz .byte_loop

```
pop  hl
ld   de, 16
add  hl, de                  ; advance to next row
pop  bc

; Move screen pointer down one character row
call next_char_row

djnz .row_loop

jr   main_loop
```

; — Data — inspect_addr: dw $C000 ; start address to inspect current_bank: db 0 ; current bank at $C000 bank_shadow: db 0 ; shadow of port $7FFD

; read_keys, print_hex, print_char, clear_screen, ; draw_header, next_char_row, bank_switch: implementations ; omitted for brevity – see examples/mem_inspect.a80 ; for the complete compilable source.

```
The key architectural point: we inspect `$C000` because that is the banked slot. By changing `

### Agon Version

The Agon version uses MOS system calls for keyboard input and VDP text output. No screen memor

```z80 id:ch15_agon_version
; Memory Inspector - Agon Light 2 (ADL mode)
    .ASSUME ADL=1
    ORG  $040000

main_loop:
    ; Wait for vsync via MOS sysvar
    rst  $08
    db   $08                    ; mos_sysvars
    ld   a, (ix+$00)            ; sysvar_time (low byte)
.wait_vsync:
    cp   (ix+$00)
    jr   z, .wait_vsync         ; spin until counter changes

    ; Check keyboard (Q = up, A = down)
    rst  $08
    db   $00                    ; mos_getkey
    ; ... navigation same as Spectrum version ...
```
```

```
.draw:
    rst  $10
    db   30                     ; VDU 30 = cursor home

    ld   hl, (inspect_addr)     ; 24-bit load!
    ld   b, 16
.row_loop:
    push bc
    push hl
    call print_hex24            ; print full 24-bit address
    ld   a, ':'
    rst  $10
    pop  hl
    push hl
    ld   b, 16
.byte_loop:
    ld   a, (hl)                ; direct 24-bit access, no banking
    call print_hex8
    inc  hl
    djnz .byte_loop
    pop  hl
    ld   de, 16
    add  hl, de
    pop  bc
    djnz .row_loop
    jr   main_loop

inspect_addr: dl $000000        ; 24-bit address (dl, not dw)
; Full source: examples/mem_inspect_agon.a80
```

Notice the contrast:

- **No bank switching.** The Agon inspector can look at any address in 512 KB directly. `LD HL,$070000` and you are inspecting 448 KB into RAM. No ports, no shadow variables, no risk of banking in the wrong page.
- **No screen address calculation.** Text output goes through `RST $10`, and the VDP handles cursor positioning, character rendering, and scrolling.
- **24-bit data directives.** We use `dl` (define long) for 3-byte pointers instead of `dw` (define word).
- **VSync through system variables.** MOS provides a `sysvar_time` counter that increments each frame. We spin-wait on it for frame synchronisation – cruder than the Spectrum's `HALT`, but functional.

Both inspectors do the same job. The Spectrum version is more code (you must handle everything yourself) but gives you total control. The Agon version is less code (the OS and VDP handle display) but gives you less control over exactly how the output looks.

This mirrors the broader development experience on both platforms. The Spectrum demands more effort for less visual richness. The Agon demands less effort for more visual richness. Both reward understanding the hardware.

Summary

- The **ZX Spectrum 128K** has 128 KB of RAM in 8 pages of 16 KB. Pages 2 and 5 are fixed in the address space; the top 16 KB slot at `$C000` is switchable via port `$7FFD`. Keep your main code in page 2 and your interrupt handler away from banked memory.
- **Contended memory** slows CPU access to certain RAM pages during active display on original Sinclair hardware. Average penalty: ~0.92 extra T-states per byte. Pentagon clones have no contention. For game development, budget a 15-20% overhead on screen writes and keep time-critical code in uncontended pages.
- **ULA timing:** the interrupt fires at the top of the frame. You get ~14,000 T-states of contention-free time before the beam enters the active display area. Use this window for screen writes.
- The **$7FFD port** is write-only. Shadow its value in RAM. Bit 3 selects the shadow screen (page 7) for double buffering. Bit 5 disables paging permanently until reset.
- **Floating bus**, **ULA snow**, and the **$7FFD read bug** are quirks of original Sinclair hardware. Avoid I register values `$40`-`$7F`. Do not store data at address `$7FFD`. The floating bus is not present on clones.
- **Pentagon 128**: no contended memory, 71,680 T-states per frame, 320 scanlines. The demoscene standard. 7 MHz turbo mode doubles the frame budget on some variants.
- **Scorpion ZS-256**: 256 KB RAM (16 pages), GMX 320x200x16 colour mode.
- **ZX Spectrum Next**: 1-2 MB RAM with 8 KB MMU pages, Layer 2 (256-colour bitmap), 128 hardware sprites, Copper co-processor, zxnDMA, triple AY sound.
- The **Agon Light 2** uses a dual-processor architecture: eZ80 @ 18.432 MHz for logic, ESP32 for video/audio. 512 KB flat RAM, 24-bit addressing (ADL mode), MOS API for system services, VDP commands for all graphics.
- **ADL mode vs Z80 mode**: ADL mode uses 24-bit registers and addresses. Z80 mode emulates classic Z80 with 16-bit addresses via MBASE. Stay in ADL mode for new Agon code.
- The **serial link** between eZ80 and ESP32 is the Agon's bottleneck. Minimise VDP command traffic per frame. Use hardware sprites and tilemaps to reduce the number of drawing commands.
- Both platforms reward careful frame budget management. The Spectrum gives you ~70,000 T-states and demands byte-level optimisation. The Agon gives you ~307,000 T-states (at 60 Hz) but throttles graphics through a serial link. Different constraints, same discipline.

Sources: Introspec "GO WEST Parts 1–2" (Hype 2015); ZX Spectrum 128K Service Manual; Zilog eZ80 CPU User Manual; Agon Light 2 Documentation (Bernardo Kastrup); ZX Spectrum Next User Manual (2nd Edition)

Chapter 16: Fast Sprites

“Two colours per cell? Fine. But those two colours are going to move.”

Every game needs things that move. Bullets, enemies, the player character, explosions. On any hardware with a blitter or a GPU, the mechanics of putting a small image at an arbitrary screen position are handled for you. On the ZX Spectrum, they are your problem.

The Spectrum has no hardware sprites, no blitter, no co-processor. Every pixel of every sprite is placed by your Z80 code, one instruction at a time, into the same video memory the ULA is reading 50 times per second. And because the screen memory layout is interleaved (Chapter 2), “one row down” means `INC H` — unless you are crossing a character boundary, in which case it means something considerably uglier.

This chapter presents six methods for drawing sprites on the Spectrum, from the simplest XOR routine to compiled sprites that execute at the theoretical maximum speed of the hardware. Each method has trade-offs. We will also look at the Agon Light 2, where the VDP provides hardware sprites and the entire problem collapses to a handful of API calls.

Method 1: XOR Sprites

The simplest approach

XOR drawing is the minimum viable sprite. It requires no mask data, no background save buffer, and no erase step. You draw the sprite by XORing its pixel data with the screen, and you erase it by XORing the same data again — the property of XOR that `A XOR B XOR B = A` guarantees that the background is restored.

Here is a complete 16x16 XOR sprite routine:

“‘z80 id:ch16_the_simplest_approach ; Draw a 16x16 XOR sprite ; Input: HL = screen address (top-left byte of sprite position) ; IX = pointer to sprite data (32 bytes: 2 bytes x 16 rows) ; xor_sprite_16x16: ld b, 16 ; 7 T 16 rows

.row: ld a, (ix+0) ; 19 T left byte of sprite row xor (hl) ; 7 T combine with screen ld (hl), a ; 7 T write back inc l ; 4 T move right one byte

```
ld   a, (ix+1)          ; 19 T   right byte of sprite row
xor  (hl)               ;  7 T
ld   (hl), a            ;  7 T   write back
```

```
dec  l                  ;  4 T   restore column

inc  ix                 ; 10 T   \  advance sprite
inc  ix                 ; 10 T   /  data pointer

inc  h                  ;  4 T   move down one pixel row
ld   a, h               ;  4 T   \
and  7                  ;  7 T    | check character
jr   nz, .no_boundary   ; 12/7T  /  boundary crossing

; Character boundary: adjust HL
ld   a, l               ;  4 T
add  a, 32              ;  7 T
ld   l, a               ;  4 T
jr   c, .no_fix         ; 12/7T
ld   a, h               ;  4 T
sub  8                  ;  7 T
ld   h, a               ;  4 T
```

.no_fix:

.no_boundary: djnz .row ; 13 T (8 on last) ret ; 10 T

```
The inner loop costs 134 T-states per row in the common case (no boundary crossing): two IX lo
and-store sequences at 18T each, two INC IX at 10T, row advance (INC H + check) at 27T, and DJ
states (the extra adjustment instructions replace the taken JR). For 16 rows: approximately 2,
states per draw. To erase the sprite, call the same routine again with the same screen address

Total cost to animate one XOR sprite: ~4,400 T-states per frame (draw + erase).

### When XOR works

XOR sprites are perfect for:

- **Cursors.** A blinking text cursor, a crosshair, a selection highlight. Anything that sits
- **Bullets.** A 2x2 or 4x4 projectile that moves fast enough that visual glitches are invisib
- **Debug markers.** Plotting collision boxes, entity positions, path nodes.

### When XOR fails

XOR has two serious problems. First, the visual quality is poor. Where the sprite overlaps wit

Second, XOR gives you no control over attributes. The sprite's colour is whatever ink/paper co

Despite its limitations, XOR is useful enough that every game programmer should have it in the

---

## Method 2: OR+AND Masked Sprites

### The industry standard
```

Almost every commercial Spectrum game released after 1984 used this technique or a close varia

The drawing algorithm for each byte is:

1. Read the screen byte.
2. AND it with the mask. This clears the pixels where the sprite will appear, leaving the rest
3. OR it with the graphic. This stamps the sprite's pixels into the cleared area.

The result: the sprite appears on screen with transparent areas showing the background through
looking sprites.

Data format

For a 16x16 sprite, each row contains 4 bytes: mask-left, graphic-left, mask-
right, graphic-right. The mask byte has `1` for transparent pixels and `0` for opaque pixels (

The inner loop

```z80 id:ch16_the_inner_loop
; Draw a 16x16 masked sprite (byte-aligned)
; Input:  HL = screen address
;         DE = pointer to sprite data
;              Format per row: mask_L, gfx_L, mask_R, gfx_R
;
masked_sprite_16x16:
    ld   b, 16              ;  7 T

.row:
    ; --- Left byte ---
    ld   a, (de)            ;  7 T   load mask
    and  (hl)               ;  7 T   clear sprite-shaped hole in background
    inc  de                 ;  6 T
    ld   c, a               ;  4 T   save masked background

    ld   a, (de)            ;  7 T   load graphic
    or   c                  ;  4 T   stamp sprite into hole
    ld   (hl), a            ;  7 T   write to screen
    inc  de                 ;  6 T
    inc  l                  ;  4 T   move right

    ; --- Right byte ---
    ld   a, (de)            ;  7 T   load mask
    and  (hl)               ;  7 T
    inc  de                 ;  6 T
    ld   c, a               ;  4 T

    ld   a, (de)            ;  7 T   load graphic
    or   c                  ;  4 T
    ld   (hl), a            ;  7 T
    inc  de                 ;  6 T
    dec  l                  ;  4 T   restore column
```

```
    ; --- Next row (DOWN_HL) ---
    inc  h                  ;  4 T
    ld   a, h               ;  4 T
    and  7                  ;  7 T
    jr   nz, .no_boundary   ; 12/7T

    ld   a, l               ;  4 T
    add  a, 32              ;  7 T
    ld   l, a               ;  4 T
    jr   c, .no_fix         ; 12/7T
    ld   a, h               ;  4 T
    sub  8                  ;  7 T
    ld   h, a               ;  4 T
.no_fix:
.no_boundary:
    djnz .row               ; 13 T
    ret                     ; 10 T
```

Cycle counting

Let us count the common case (no character boundary crossing). Note that `JR NZ` is taken (12T) in the common case because the boundary crossing is rare — only 1 in 8 rows crosses a character boundary.

Section	Instructions	T-states
Left byte: mask+draw	`ld a,(de)` + `and (hl)` + `inc de` + `ld c,a` + `ld a,(de)` + `or c` + `ld (hl),a` + `inc de` + `inc l`	52
Right byte: mask+draw	Same sequence + `dec l`	52
Row advance	`inc h` + `ld a,h` + `and 7` + `jr nz` (taken)	27
Loop	`djnz`	13
Total per row		**144**

For 16 rows: 16 x 144 = **2,304 T-states** (common case). Add boundary-crossing overhead for ~2 boundaries in a 16-pixel sprite: roughly **2,400 T-states** total.

But this only draws the sprite. You also need to erase the previous frame's sprite, which means restoring the background — we will address this in Method 6 (Dirty Rectangles). For now, note that the draw alone is about 35% more expensive than XOR, but the visual quality is incomparably better.

Byte alignment and the shift problem

The routine above assumes the sprite starts at a byte boundary — that is, the x coordinate is a multiple of 8. In practice, game characters move pixel by pixel, not in 8-pixel jumps. If your sprite's x position is 53, it starts at byte column 6, pixel 5 within that byte. The sprite data needs to be shifted right by 5 bits.

You can shift at draw time:

```
z80 id:ch16_byte_alignment_and_the_shift ; Shift mask and graphic right by A
bits ; This adds significant cost per byte     ld   a, (de)            ;  7 T
load mask byte     ld   c, a              ;  4 T     ld   a, b              ;
4 T   shift count .shift:     srl  c                  ;  8 T   \     dec  a
;  4 T    | per-bit shift loop     jr   nz, .shift         ; 12 T   /
```

Each bit of shift costs 24 T-states per byte in this naive loop. For a 5-bit shift on a 16-wide sprite (3 bytes per row after shift, since the sprite spills into a third byte), you are looking at an extra 5 x 24 x 3 = 360 T-states per row — doubling the draw cost. For 8 sprites at 25 fps, this shift overhead alone would consume roughly 46,000 T-states per frame — over 60% of your budget.

This is why pre-shifted sprites exist.

Method 3: Pre-Shifted Sprites

The memory-for-speed trade-off

The idea is simple: instead of shifting the sprite data at draw time, pre-compute shifted versions of the sprite at load time (or at assembly time) and store them alongside the original. When you need to draw the sprite at pixel offset 3 within a byte, you use the version that was pre-shifted by 3 pixels.

There are two common configurations:

4 shifted copies (shift by 0, 2, 4, 6 pixels). This gives 2-pixel horizontal resolution. The sprite snaps to even pixel positions, which is often acceptable for game characters. Memory cost: 4x the unshifted data.

8 shifted copies (shift by 0, 1, 2, 3, 4, 5, 6, 7 pixels). Full pixel-level horizontal positioning. Memory cost: 8x the unshifted data. But each shifted version is also wider: a 16-pixel-wide sprite shifted by 1–7 bits spills into a third byte column, so each shifted copy is 3 bytes wide instead of 2.

Memory calculation

For a 16x16 masked sprite:

Configuration	Bytes per row	Rows	Copies	Total
Unshifted only	4 (mask+gfx x 2 bytes)	16	1	64
4 shifts	4	16	4	256
8 shifts (3 bytes wide)	6 (mask+gfx x 3 bytes)	16	8	768

For a sprite with 4 animation frames, multiply by 4:

Configuration	Per frame	4 frames	8 sprites
Unshifted + runtime shift	64	256	2,048
4 pre-shifts	256	1,024	8,192

Configuration	Per frame	4 frames	8 sprites
8 pre-shifts	768	3,072	24,576

24 KB for 8 sprites with full pre-shifting. On a 128K Spectrum, that is one and a half memory banks just for sprite data. On a 48K machine, it is nearly half of available RAM. The trade-off is stark.

Practical compromise

Most games use 4 pre-shifted copies. The 2-pixel horizontal resolution is barely noticeable in gameplay. Some games use 8 copies for the player character (where smooth movement matters most) and 4 copies or even runtime shifting for less important sprites.

The draw routine for pre-shifted sprites is identical to the byte-aligned masked sprite routine — you just select the correct pre-shifted data set before calling it:

```
z80 id:ch16_practical_compromise ; Select pre-shifted sprite data ; Input:  A
= x coordinate (0-255) ;         IY = base of pre-shift table (4 entries, each
pointing to 16-row data) ; Output: DE = pointer to correct shifted sprite data
; select_preshift:     and  $06                ;  7 T   mask to shifts 0,2,4,6
(4 copies)     ld   c, a               ;  4 T     ld   b, 0               ;  7
T     add  iy, bc             ; 15 T     ld   e, (iy+0)          ; 19 T     ld
d, (iy+1)          ; 19 T   DE = pointer to shifted data     ret
```

The draw time is the same as Method 2: ~2,300 T-states. But you have eliminated the per-pixel shift cost entirely. The price is paid in memory, not T-states.

Method 4: Stack Sprites (The PUSH Method)

The fastest output on the Z80

We saw in Chapter 3 that PUSH writes 2 bytes in 11 T-states — 5.5 T-states per byte, the fastest write operation on the Z80. Stack sprites exploit this for sprite output: set SP to the bottom of the sprite's screen area, load register pairs with sprite data, and PUSH them onto the screen.

The technique requires a critical setup:

1. **DI** — disable interrupts. If an interrupt fires while SP points to the screen, the CPU pushes its return address into your pixel data, corrupting the display and probably crashing.
2. **Save SP** — stash the real stack pointer using self-modifying code.
3. **Set SP** to the bottom-right of the sprite area on screen (PUSH works downward).
4. **Load and PUSH** — load sprite data into register pairs and PUSH them in sequence.
5. **Restore SP and EI** — put the stack back and re-enable interrupts.

The inner loop

For a 16x16 sprite (2 bytes wide), each row is a single PUSH:

“‘z80 id:ch16_the_inner_loop_2 ; Stack sprite: 16x16, writes 2 bytes per row via PUSH ; Input: screen_addr = pre-calculated bottom-right screen address ; sprite_data = 32 bytes of pixel data (2 bytes x 16 rows, ; stored bottom-to-top because PUSH goes downward) ; stack_sprite_16x16: di ; 4 T

```
ld   (restore_sp + 1), sp    ; 20 T   save SP (self-mod)

ld   sp, (screen_addr)       ; 20 T   SP = bottom of sprite on screen
ld   ix, sprite_data         ; 14 T

; Row 15 (bottom) - each PUSH writes 2 bytes and decrements SP
ld   h, (ix+31)              ; 19 T   \
ld   l, (ix+30)              ; 19 T    | load bottom row
push hl                      ; 11 T   /  write to screen

; But wait --- SP just decremented by 2, and the next row UP
; on the Spectrum screen is NOT at SP-2. The interleaved layout
; means "one row up" is at a completely different address.
;
; This is the fundamental problem with stack sprites on the
; Spectrum: the screen is not contiguous in memory.


And here is the fundamental difficulty. The PUSH method is the fastest possible write, but the

### Making it work: pre-calculated SP chain

The solution is to not rely on SP auto-decrement for row navigation. Instead, you explicitly s

```z80 id:ch16_making_it_work_pre_calculated
; Stack sprite: 16x16 with explicit SP per row
; This is the practical version --- each row gets SP set independently
;
stack_sprite_16x16:
    di                           ;  4 T
    ld   (restore_sp + 1), sp    ; 20 T

    ld   hl, (sprite_data + 0)   ; 16 T   row 0 data
    ld   sp, (row_addrs + 0)     ; 20 T   SP = screen addr for row 0 + 2
    push hl                      ; 11 T   write row 0
                                 ;        total per row: 47 T

    ld   hl, (sprite_data + 2)   ; 16 T   row 1 data
    ld   sp, (row_addrs + 2)     ; 20 T
    push hl                      ; 11 T
    ; ... repeated for all 16 rows ...

restore_sp:
    ld   sp, $0000               ; 10 T   self-modified
```
```

```
    ei                          ;  4 T
    ret                         ; 10 T
```

Each row costs 47 T-states. For 16 rows: 16 x 47 = 752 T-states, plus setup and teardown (~60 T-states). Total: approximately **810 T-states**.

Compare this to Method 2's ~2,300 T-states. The stack sprite is nearly 3x faster — but it comes with constraints.

The costs

No masking. The PUSH writes 2 bytes unconditionally. It overwrites whatever was on screen. There is no AND-with-mask step. The sprite is always a solid rectangle — any "transparent" pixels will show the sprite's background colour, not the game background. For sprites on a solid-colour background (many classic Spectrum games used black), this is fine. For sprites over detailed backgrounds, it is not.

Pre-calculated row addresses. You need a table of screen addresses for all 16 rows of the sprite, updated whenever the sprite moves. This is a per-frame setup cost — not enormous, but not free.

Interrupts are off. For 8 sprites, roughly 6,500 T-states with interrupts disabled. If your music runs from IM2, schedule sprite drawing immediately after HALT.

Data must be stored in PUSH order. Since PUSH writes the high byte at (SP-1) and the low byte at (SP-2), and SP decrements *before* writing, the data layout requires careful attention. The sprite data is stored reversed: the rightmost byte of a row becomes the low byte loaded into the register, the leftmost becomes the high byte.

When to use stack sprites

Stack sprites are the weapon of choice when you need raw speed and your background is simple enough that full-rectangle overwrites are acceptable. Arcade-style games with black backgrounds, score overlays, and fast-moving objects are the natural fit. The PUSH method is also used for screen clearing and bulk data output (Chapter 3), where the "no masking" limitation is irrelevant.

Method 5: Compiled Sprites

The sprite is the code

A compiled sprite takes the code-generation philosophy from Chapter 3 to its logical conclusion. Instead of a data table interpreted by a draw routine, the sprite *is* an executable routine. Each visible pixel byte in the sprite becomes a `LD (HL), n` instruction. Transparent runs become `INC L` or `INC H` instructions to skip over them. The entire sprite is rendered by `CALL`-ing it.

How it works

Consider a simple 8x8 sprite with some transparent pixels. In a masked sprite, you would store mask+graphic pairs and run the AND/OR loop. In a compiled sprite,

you generate Z80 instructions at assembly time (or at load time):

“‘z80 id:ch16_how_it_works ; Compiled sprite for a small arrow shape ; Input: HL = screen address of top-left byte ; The sprite draws itself. ; compiled_arrow: ; Row 0: one pixel byte ld (hl), $18 ; 10 T ..##.... inc h ; 4 T next row

```
; Row 1: one pixel byte
ld   (hl), $3C          ; 10 T   ..####..
inc  h                  ;  4 T

; Row 2: one pixel byte
ld   (hl), $7E          ; 10 T   .######.
inc  h                  ;  4 T

; Row 3: one pixel byte
ld   (hl), $FF          ; 10 T   ########
inc  h                  ;  4 T

; Row 4: one pixel byte
ld   (hl), $3C          ; 10 T   ..####..
inc  h                  ;  4 T

; Row 5: one pixel byte
ld   (hl), $3C          ; 10 T   ..####..
inc  h                  ;  4 T

; Row 6: one pixel byte
ld   (hl), $3C          ; 10 T   ..####..
inc  h                  ;  4 T

; Row 7: one pixel byte
ld   (hl), $3C          ; 10 T   ..####..

ret                     ; 10 T

; Total: 8 x (10 + 4) + 10 = 122 T-states
; Compare: masked routine for 8x8 = ~600 T-states


That is 122 T-states for an 8x8 sprite. The masked approach takes roughly 5x longer.

### 16x16 compiled sprite

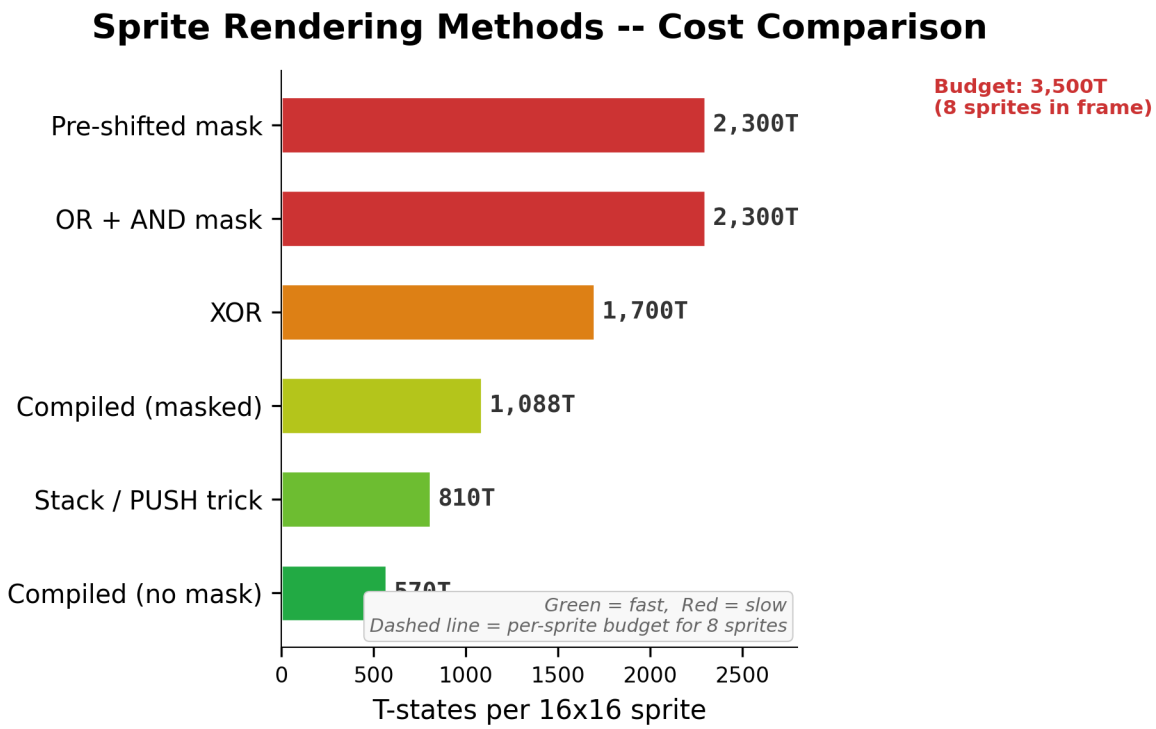
For a wider sprite, each row may have multiple `LD (HL), n` instructions separated by `INC L`:

```z80 id:ch16_16x16_compiled_sprite
; Compiled sprite: 16x16 (2 bytes wide)
; Input:  HL = screen address of top-left
;
compiled_sprite_16x16:
    ; Row 0
    ld   (hl), $3C          ; 10 T   left byte
    inc  l                  ;  4 T
```
```

```
    ld   (hl), $0F          ; 10 T   right byte
    dec  l                  ;  4 T   restore column
    inc  h                  ;  4 T   next row
                            ;        row cost: 32 T

    ; Row 1
    ld   (hl), $7E          ; 10 T
    inc  l                  ;  4 T
    ld   (hl), $1F          ; 10 T
    dec  l                  ;  4 T
    inc  h                  ;  4 T
                            ;        row cost: 32 T

    ; ... rows 2-6 similar ...

    ; Row 7 (character boundary)
    ld   (hl), $FF          ; 10 T
    inc  l                  ;  4 T
    ld   (hl), $FF          ; 10 T
    dec  l                  ;  4 T
    ; Character boundary crossing:
    ld   a, l               ;  4 T
    add  a, 32              ;  7 T
    ld   l, a               ;  4 T
    ld   a, h               ;  4 T
    sub  8                  ;  7 T
    ld   h, a               ;  4 T   boundary cost: 30 T
    inc  h                  ;  4 T
                            ;        row cost: 62 T

    ; Rows 8-15 similar to 0-6, with another boundary at row 15
    ; ...
    ret                     ; 10 T
```

Per row (common case): 32 T-states. For 16 rows with 1–2 boundary crossings: roughly **570 T-states**.

The trade-offs

Strengths: - The fastest masked-capable sprite method. You can incorporate AND masking into compiled sprites — each byte becomes `ld a,(hl)` / `and mask` / `or graphic` / `ld (hl),a` instead of a simple `ld (hl),n`. Even with masking, the compiled approach avoids loop overhead, data pointer management, and row counting. - No loop overhead at all. The code is completely straight-line. - Transparent regions cost nothing if they span entire bytes — you just skip them with `INC L` or `INC H`.

Weaknesses: - **Code size.** Each visible byte takes 2 bytes of code (`LD (HL), n`). With masking (4 instructions per byte), code size roughly triples. A full set of 8 pre-shifted compiled sprites with 4 animation frames can reach several kilobytes per sprite. - **No runtime data changes.** Pixel values are baked into instruction operands. Animation requires a separate compiled routine per frame. - **Boundary handling is baked in.** Character boundary crossings sit at fixed positions, so the

sprite must maintain consistent vertical alignment or you need multiple compiled versions.

Compiled sprites with masking

For sprites that need to appear over a detailed background, you compile the mask into the code:

```
z80 id:ch16_compiled_sprites_with_masking ; Compiled sprite with masking: one
byte ; Instead of ld (hl),n, we do:     ld   a, (hl)            ;  7 T   read
screen     and  $C3                ;  7 T   mask: clear sprite pixels     or
$3C                ;  7 T   graphic: stamp sprite     ld   (hl), a            ;
7 T   write back                            ;        per-byte cost: 28 T
```

28 T-states per byte, versus 52 T-states per byte in the general-purpose masked routine (Method 2). The saving comes from eliminating pointer management, loop counting, and data loading — the mask and graphic values are immediate operands.

For 16 rows x 2 bytes: 16 x (28 + 28 + 4 + 4 + 4) = 16 x 68 = **1,088 T-states**. This is about half the cost of the generic masked routine, with full transparency support.

Method	16x16 draw cost	Masking	Notes
XOR sprite	~2,200 T	No	Draw + erase = ~4,400 T
OR+AND masked	~2,400 T	Yes	Standard approach
Pre-shifted masked	~2,400 T	Yes	No shift cost; 4–8x memory
Stack sprite (PUSH)	~810 T	No	DI required; solid rectangle
Compiled (no mask)	~570 T	No	Code = sprite; large footprint
Compiled (masked)	~1,088 T	Yes	Best of both; largest footprint

---

Method 6: Dirty Rectangles

The background problem

Methods 1–5 all address the question of *putting* pixels on screen. But sprites move. Every frame, the sprite is in a new position. Before drawing the sprite at its new position, you must erase it from the old position — or the screen fills up with ghostly after-images.

The XOR method handles this implicitly: XOR the old position to erase, XOR the new position to draw. But for all other methods, you need a way to restore the background.

There are three common approaches:

Full screen clear. Wipe the pixel area every frame (~36,000 T-states with PUSH from Chapter 3), then redraw everything. Feasible but expensive.

Background save/restore. Before drawing each sprite, save the screen behind it. To erase, copy the saved buffer back. Cost is O(sprite_size) per sprite, not O(screen_size).



Figure 11: Sprite rendering methods comparison

Dirty rectangle tracking. A refinement: track which rectangles were modified, restore only those, then draw new sprites (saving new background as you go).

The save/restore cycle

The practical approach for most Spectrum games is per-sprite background save/restore. Here is the cycle for one sprite per frame:

1. **Restore** the background saved last frame (copy saved buffer to old screen position).
2. **Save** the background at the new screen position (copy screen to save buffer).
3. **Draw** the sprite at the new position.

The order matters. You restore before saving to avoid overwriting the new background save with stale data if sprites overlap.

Save/restore routine

For a 16x16 sprite (2 bytes wide, 16 rows), the background buffer is 32 bytes:

“‘z80 id:ch16_save_restore_routine ; Save background behind a 16x16 sprite ; Input: HL = screen address (top-left) ; DE = pointer to save buffer (32 bytes) ; save_background_16x16: ld b, 16 ; 7 T

.row: ld a, (hl) ; 7 T read left byte ld (de), a ; 7 T save it inc de ; 6 T inc l ; 4 T

```
ld   a, (hl)           ;  7 T   read right byte
ld   (de), a           ;  7 T   save it
inc  de                ;  6 T
dec  l                 ;  4 T
```

```
; DOWN_HL (same as sprite routines)
inc  h                  ;  4 T
ld   a, h               ;  4 T
and  7                  ;  7 T
jr   nz, .no_boundary   ; 12/7T
ld   a, l               ;  4 T
add  a, 32              ;  7 T
ld   l, a               ;  4 T
jr   c, .no_fix         ; 12/7T
ld   a, h               ;  4 T
sub  8                  ;  7 T
ld   h, a               ;  4 T
```

.no_fix: .no_boundary: djnz .row ; 13 T ret ; 10 T

```
The restore routine is identical with source and destination swapped: read from the buffer, wr
states** for 16 rows.

### Full frame budget

Let us calculate the per-frame cost for 8 animated 16x16 sprites using OR+AND masking with bac

| Operation | Per sprite | 8 sprites |
|-----------|-----------|-----------|
| Restore background | ~1,500 T | 12,000 T |
| Save new background | ~1,500 T | 12,000 T |
| Draw sprite (masked) | ~2,400 T | 19,200 T |
| **Total** | **~5,400 T** | **~43,200 T** |

On a Pentagon (71,680 T-states per frame): 43,200 T leaves **28,480 T** for game logic, input
states for non-sprite work. This is comfortable for a game.

If you use compiled masked sprites instead:

| Operation | Per sprite | 8 sprites |
|-----------|-----------|-----------|
| Restore background | ~1,500 T | 12,000 T |
| Save new background | ~1,500 T | 12,000 T |
| Draw sprite (compiled, masked) | ~1,088 T | 8,704 T |
| **Total** | **~4,088 T** | **~32,704 T** |

That saves over 10,000 T-states per frame --- a meaningful improvement that buys you more room

### Draw order and overlap

When multiple sprites overlap, draw order matters. The simplest correct approach:

1. Restore all backgrounds (in reverse draw order, to handle overlaps correctly).
2. Save all new backgrounds.
3. Draw all sprites.
```

```
Reverse-order restoration ensures that when two sprites overlapped last frame, the earlier spr

The reasoning: if sprite A was drawn on top of sprite B, A's save buffer contains B's pixels.
order restoration leaves artifacts. Many games sidestep this by preventing overlap or acceptin

---

## Optimising the Inner Loops

### Eliminating pointer management

The routines above spend significant time on pointer management: `inc de`, `inc l`, `dec l`, a

**Use LDI instead of manual copy.** For save/restore operations, an LDI chain (Chapter 3) copi
states. Compared to our manual `ld a,(hl)` + `ld (de),a` + `inc de` + `inc l` at 24 T-
states, LDI saves 8 T-states per byte. For a 16x16 sprite (32 bytes), that is 256 T-
states saved per save or restore operation.

```z80 id:ch16_eliminating_pointer
; Save background using LDI (partial unroll, 2 bytes per row)
; HL = screen address, DE = save buffer
;
save_bg_ldi:
    ld   b, 16              ;  7 T
.row:
    ldi                     ; 16 T   copy left byte
    ldi                     ; 16 T   copy right byte
    dec  l                  ;  4 T   \
    dec  l                  ;  4 T   /  LDI advanced L by 2, undo it

    ; DOWN_HL
    inc  h                  ;  4 T
    ld   a, h               ;  4 T
    and  7                  ;  7 T
    jr   nz, .no_boundary   ; 12/7T
    ld   a, l               ;  4 T
    add  a, 32              ;  7 T
    ld   l, a               ;  4 T
    jr   c, .no_fix         ; 12/7T
    ld   a, h               ;  4 T
    sub  8                  ;  7 T
    ld   h, a               ;  4 T
.no_fix:
.no_boundary:
    djnz .row               ; 13 T
    ret                     ; 10 T
```

Common-case row cost: 16 + 16 + 4 + 4 + 4 + 4 + 7 + 12 + 13 = **80 T-states** (JR NZ is taken at 12T in the common case — no boundary crossing). For 16 rows: approximately **1,280 T-states** — a worthwhile improvement over the 1,500 T-states of the manual approach.
```

Combine save and draw. Instead of save-then-draw as two separate passes over the screen area, combine them into one pass: for each byte, read the screen (save it), then write the sprite data. This halves the number of row-advance operations and eliminates one complete DOWN_HL traversal:

“‘z80 id:ch16_eliminating_pointer_2 ; Combined save-and-draw for masked sprite ; HL = screen address, DE = sprite data (mask, gfx pairs) ; IX = save buffer ; save_and_draw_16x16: ld b, 16 ; 7 T .row: ; Left byte ld a, (hl) ; 7 T read screen (for saving) ld (ix+0), a ; 19 T save to buffer ld c, a ; 4 T

```
ld   a, (de)            ;  7 T   load mask
and  c                  ;  4 T   mask background
inc  de                 ;  6 T
ld   c, a               ;  4 T

ld   a, (de)            ;  7 T   load graphic
or   c                  ;  4 T   stamp sprite
ld   (hl), a            ;  7 T   write to screen
inc  de                 ;  6 T
inc  l                  ;  4 T

; Right byte (similar)
ld   a, (hl)            ;  7 T
ld   (ix+1), a          ; 19 T
ld   c, a               ;  4 T

ld   a, (de)            ;  7 T
and  c                  ;  4 T
inc  de                 ;  6 T
ld   c, a               ;  4 T

ld   a, (de)            ;  7 T
or   c                  ;  4 T
ld   (hl), a            ;  7 T
inc  de                 ;  6 T
dec  l                  ;  4 T

; Advance IX and HL
inc  ix                 ; 10 T
inc  ix                 ; 10 T

inc  h                  ;  4 T
ld   a, h               ;  4 T
and  7                  ;  7 T
jr   nz, .no_boundary   ; 12/7T
ld   a, l               ;  4 T
add  a, 32              ;  7 T
ld   l, a               ;  4 T
jr   c, .no_fix         ; 12/7T
ld   a, h               ;  4 T
sub  8                  ;  7 T
ld   h, a               ;  4 T
```

.no_fix: .no_boundary: djnz .row ; 13 T ret ; 10 T

```
This combines save and draw into a single pass. The cost per row (common case): roughly **205
states** (JR NZ taken at 12T). For 16 rows: approximately **3,400 T-states** --- compared to s
states. The saving is modest (~280 T per sprite), but it adds up across 8 sprites.

For maximum performance, unroll the entire routine: no DJNZ loop, explicit per-
row code with boundary crossings baked in at rows 7 and 15. This eliminates loop and boundary-
test overhead, bringing the total to roughly **2,780 T-states** at the cost of ~300 bytes of c

---

## Agon Light 2: Hardware VDP Sprites

The Agon Light 2 takes a fundamentally different approach. The eZ80 communicates with a VDP (V

The VDU command sequence for defining and activating a sprite:

```text
VDU 23, 27, 0, n                    ; Select sprite n
VDU 23, 27, 1, w, h, format         ; Create sprite: w x h pixels
; ... upload bitmap data ...
VDU 23, 27, 4, x_lo, x_hi, y_lo, y_hi  ; Set position
VDU 23, 27, 11                       ; Show sprite
```

In eZ80 assembly, these commands are sent as byte sequences to the VDP via `RST $10` (MOS character output). Each command is a sequence of `ld a, byte : rst $10` pairs.

Moving a sprite

Once defined, moving a sprite is just the position command:

“‘z80 id:ch16_moving_a_sprite ; Agon: Move sprite 0 to (x, y) ; Input: BC = x, DE = y ; move_sprite: ld a, 23 : rst $10 ld a, 27 : rst $10 ld a, 4 : rst $10 ; move command ld a, 0 : rst $10 ; sprite number

```
ld   a, c  : rst $10    ; x low
ld   a, b  : rst $10    ; x high
ld   a, e  : rst $10    ; y low
ld   a, d  : rst $10    ; y high
ret


The CPU cost for moving a sprite is just the cost of sending ~10 bytes over the serial interfa
states at 18.432 MHz. Moving 8 sprites: ~13,000 T-states. The VDP handles all pixel compositin

### Scanline limits

The VDP has a practical limit on sprite pixels per horizontal scanline. When too many sprites

### The trade-off
```
```

```
The Agon eliminates the entire sprite-drawing problem. No save/restore, no masking, no interle
pixel tricks, no creative data manipulation, and dependency on VDP firmware capabilities. The

---

## Practical: 8 Animated Sprites at 25 fps

### Spectrum implementation

Our target: 8 animated 16x16 sprites with background save/restore, running at 25 fps (updating

**Architecture:**

Each sprite has a data structure:

```z80 id:ch16_spectrum_implementation
; Sprite structure (12 bytes per sprite)
;
SPRITE_X        EQU 0       ; x coordinate (0-255)
SPRITE_Y        EQU 1       ; y coordinate (0-191)
SPRITE_OLD_X    EQU 2       ; previous x (for erase)
SPRITE_OLD_Y    EQU 3       ; previous y
SPRITE_FRAME    EQU 4       ; current animation frame (0-3)
SPRITE_DIR      EQU 5       ; direction / flags
SPRITE_DX       EQU 6       ; x velocity (signed)
SPRITE_DY       EQU 7       ; y velocity (signed)
SPRITE_GFX      EQU 8       ; pointer to sprite graphic data (2 bytes)
SPRITE_SAVE     EQU 10      ; pointer to background save buffer (2 bytes)

SPRITE_SIZE     EQU 12
NUM_SPRITES     EQU 8
```

Frame cycle (every 2 VBLANKs):

“‘z80 id:ch16_spectrum_implementation_2 main_loop: halt ; wait for VBLANK halt ; wait again (25 fps = every 2nd frame)

```
; Phase 1: Restore all backgrounds (reverse order)
ld   ix, sprites + (NUM_SPRITES - 1) * SPRITE_SIZE
ld   b, NUM_SPRITES
```

.restore_loop: call restore_sprite_bg ld de, -SPRITE_SIZE add ix, de djnz .restore_loop

```
; Phase 2: Update positions
ld   ix, sprites
ld   b, NUM_SPRITES
```

.update_loop: call update_sprite_position ld de, SPRITE_SIZE add ix, de djnz .update_loop

```
; Phase 3: Save backgrounds and draw (forward order)
ld   ix, sprites
ld   b, NUM_SPRITES
```
```

.draw_loop: call save_and_draw_sprite ld de, SPRITE_SIZE add ix, de djnz .draw_loop

```
; Phase 4: Game logic, input, sound
call process_input
call update_game_logic
call update_sound

jr   main_loop
```

Cycle budget:

Phase	Cost
2 x HALT	0 T (waiting)
Restore 8 backgrounds	8 x 1,280 = 10,240 T
Update 8 positions	8 x 200 = 1,600 T
Save + Draw 8 sprites	8 x 3,400 = 27,200 T
Loop overhead	~2,000 T
Total sprite work	**~41,040 T**
Available for game logic	**~102,000 T**

With a 2-frame budget of 143,360 T-states (2 x 71,680 on Pentagon), we have roughly 102,000 T-states for game logic, input, and sound. This is generous --- enough for entity AI (Chapter 19

Before drawing each sprite, calculate the screen address from (x, y) using the routine from Ch shifted data based on `x AND $06` (for 4 shift levels). The pre-shift selection logic from Met

Agon implementation

On the Agon, the main loop becomes trivially simple: wait for VSync, update positions, send `V

The contrast is instructive. On the Spectrum, sprite rendering is the dominant cost --- over 4 states per frame, where every cycle saved in the inner loop translates directly to more sprite pushing. Both approaches have their satisfactions.

Summary

- **XOR sprites** are the simplest method: XOR to draw, XOR again to erase. ~2,200 T-states to draw a 16x16 sprite. No mask, no background save needed. Visual quality is poor (inv

- **OR+AND masked sprites** are the industry standard. Each byte goes through an AND-with-mask, OR-with-graphic sequence that produces clean transparency. ~2,400 T-states for a 16x16 sprite. This is what most commercial Spectrum games use.

- **Pre-shifted sprites** eliminate the per-pixel shift cost by storing 4 or 8 pre-computed shifted copies of the sprite data. Draw time is the same as the masked routine. Memor pixel resolution) to 8x (8 shifts, full pixel resolution). The standard memory-vs-speed trade-off.

- **Stack sprites (PUSH method)** are the fastest raw output: ~810 T-states for a 16x16 sprite

- **Compiled sprites** turn the sprite into executable code. Each pixel byte becomes a `LD (HL
states without masking, ~1,088 T-states with compiled masking. The fastest masked method, at t

- **Dirty rectangles** with background save/restore are the standard technique for sprite anim
and-draw approach reduces per-sprite cost to ~3,400 T-states.

- **8 sprites at 25 fps** on a Spectrum 128K costs approximately 41,000 T-
states per update cycle (every 2 frames), leaving ~102,000 T-states for game logic --- a comfo

- **Agon Light 2 hardware sprites** eliminate the entire rendering problem. Define sprites onc
off is abstraction: you gain performance but lose the ability to do per-pixel tricks with spri

- The choice of sprite method depends on your game's requirements: background complexity, spri
shifting and dirty rectangles. Demoscene productions and performance-critical games reach for

---

Try It Yourself

1. **Implement all six methods.** Take a simple 8x8 sprite design and implement XOR, masked, a
colour timing harness from Chapter 1 to compare their draw costs. The difference should be cle

2. **Pre-shift generator.** Write a utility (in Python, Processing, or Z80 assembly) that take
shifted versions. Store them in memory and write a draw routine that selects the correct versi
coordinate.

3. **Background save/restore demo.** Put a masked sprite on a patterned background (the checke

4. **The 8-sprite challenge.** Implement the full 8-sprite system with background save/restore

5. **Agon comparison.** If you have an Agon Light 2, implement the same 8-
sprite animation using VDP hardware sprites. Compare the code complexity and the CPU budget av

---

> **Sources:** Spectrum graphics programming folklore, widely documented across the ZX communi
modifying code; Chapter 2 for screen layout and DOWN_HL; Agon Light 2 VDP documentation (Quark
dev chapters of *book-plan.md* for the six-method framework and practical targets.

\newpage

Chapter 17: Scrolling

> "The screen is 256 pixels wide. The level is 8,000. Somehow, the player must move through it

---

```
Every side-scrolling game needs to move the world. The player runs right, the background shift

This chapter works through every practical scrolling method on the Spectrum, from the cheapest
state, build comparison tables, and show how the shadow screen trick on the 128K makes the who

Then we will look at how the Agon Light 2 handles the same problem with hardware scroll offset

---

## The Budget

Before we write a single instruction, let us establish what we are working with.

On a Pentagon (the timing model most Spectrum demos and games target), one frame is **71,680 T
states**. On a standard 48K/128K Spectrum, it is 69,888. We will use the Pentagon figure throu

A full-screen scroll means moving data across all 6,144 bytes of pixel memory (and possibly th
states are left for everything else -- game logic, sprite drawing, music, input?

Here is the raw cost of just *touching* every byte in the pixel area, using different methods:

| Method | Per byte | 6,144 bytes | % of frame |
|--------|----------|-------------|------------|
| `ldir` | 21 T | 129,019 T | 180% |
| `ldi` chain | 16 T | 98,304 T | 137% |
| `ld a,(hl)` + `ld (de),a` + `inc hl` + `inc de` | 24 T | 147,456 T | 206% |
| `push` (2 bytes) | 5.5 T/byte | 33,792 T | 47% |

The first three methods cannot move the full pixel area in a single frame. Even `ldi` chains,
byte cost.

This is why scrolling on the Spectrum is a design problem, not just a coding problem. You cann
force a full-screen pixel scroll at 50fps. You must choose your method based on what your game

---

## Vertical Pixel Scrolling

Vertical scrolling moves the screen contents up or down by one or more pixel rows. Conceptuall

### The Interleave Problem

Recall the screen address structure from Chapter 2:

```text
High byte:  0 1 0 T T S S S
Low byte:   L L L C C C C C
```

Where TT = third (0–2), SSS = scanline within character cell (0–7), LLL = character
row within third (0–7), CCCCC = column byte (0–31).

To scroll up by one pixel, you need to copy the contents of row N to row N-1, for
```

every row from 1 to 191. The source and destination addresses for adjacent pixel rows are *not* separated by a constant offset. Within a character cell, consecutive rows differ by $0100 in the high byte (just `INC H` / `DEC H`). But at character cell boundaries – every 8th row – the relationship changes: you must adjust L by 32 and reset the scanline bits in H. At third boundaries (every 64th row), the adjustment is different again.

The Algorithm: Scroll Up by One Pixel

The approach that works with the interleave, rather than against it, uses the split-counter structure from Chapter 2. Maintain two pointers (source and destination) and advance both using the screen's natural hierarchy: 3 thirds, 8 character rows per third, 8 scanlines per character row. Within each character cell, moving between scanlines is just `INC H` / `INC D` on the source and destination. At character row boundaries, reset the scanline bits and add 32 to L. At third boundaries, add 8 to H and reset L. The inner loop copies 32 bytes per row with LDIR or an LDI chain, and the pointer advancement is folded into the outer loop structure.

Cost Analysis

For each of the 191 row copies, we must copy 32 bytes from source to destination. Using LDIR:

- Per row: 32 bytes x 21 T-states - 5 = 667 T-states for the LDIR, plus pointer management overhead.
- Pointer management (save/restore source and dest, advance scanline): approximately 60 T-states per row within a character cell, more at boundaries.

Total with LDIR: approximately 143,000 T-states. That is roughly **two full frames**. A vertical pixel scroll by one row using LDIR does not fit in a single frame.

We can do better. Replace the LDIR with an LDI chain – 32 LDI instructions per row:

- Per row: 32 x 16 = 512 T-states for the LDIs, plus ~50 T-states of pointer management.
- Total: 191 x 562 = **107,342 T-states.** Still over budget by about 50%.

The PUSH trick is awkward here because we need to copy *between* two non-contiguous areas with a non-constant relationship. PUSH writes to contiguous descending addresses, which does not match the interleaved source/destination pattern.

Partial Scrolling: The Practical Approach

The reality is that most games do not scroll the entire 192-line display. A typical game reserves:

- Top 2 character rows (16 pixels) for a status bar – not scrolled.
- Bottom 1 character row (8 pixels) for a score line – not scrolled.
- Middle: 21 character rows = 168 pixel rows = the scrolling play area.

168 rows of vertical pixel scrolling with LDI chains: 168 x 562 = **94,416 T-states**, or 132% of a frame. Still too much for a single frame if you want time for anything else.

This is why pure vertical pixel scrolling at 1px/frame is rare in Spectrum games. The common approaches are:

1. **Scroll by 8 pixels (one character row):** Move the attributes and the character-aligned pixel data. This is much cheaper because you only copy 21 character rows x 8 scanlines = 168 rows, but you can use a block copy trick: within each third, the character rows are contiguous in blocks. Cost: around 40,000–50,000 T-states with LDIR. Feasible.

2. **Scroll by 1 pixel using a counter:** Scroll 1px per frame visually by combining a character-level scroll (cheap, every 8 frames) with a pixel offset counter (draw new content at an offset within the 8px character cell). We will cover this combined approach in the horizontal scrolling section below, because it is far more commonly needed there.

3. **Use the shadow screen (128K only):** Draw the scrolled content into a back buffer, then flip. This eliminates tearing and lets you spread the work across frames. We cover this later in the chapter.

Scrolling by 8 Pixels (One Character Row)

Scrolling by a full character row is dramatically cheaper because the source and destination are related by a simple offset within each third. The character rows within a third are spaced 32 bytes apart in L. So scrolling one character row up means copying from L+32 to L, for each scanline and each third.

For scrolling the play area by one character row, the key insight is that within a single scanline, character rows are stored contiguously (32 bytes apart). Scanline 0 of char rows 0–7 in a third lives at `$xx00`, `$xx20`, `$xx40`, …, `$xxE0`. Scrolling N character rows up by one position within a single scanline is therefore a single block copy of (N-1) x 32 bytes.

For a 20-character-row play area, one scanline's worth of data is 20 x 32 = 640 bytes. Scrolling that scanline means copying 19 x 32 = 608 bytes forward by 32. We do this for each of the 8 scanlines, handling third boundaries separately.

Estimated cost: 8 scanlines x ~12,700 T-states per scanline (608 bytes via LDIR) + third-boundary handling = approximately **105,000 T-states**. That is 146% of a frame.

Even character-row scrolling the full play area in one frame is tight. Games handle this by either:

- **Scrolling during the blank (border) period.** The top and bottom border on a Pentagon give approximately 14,000 T-states of free time where no contention occurs.
- **Splitting across two frames.** Scroll the top half one frame, the bottom half the next. The visual effect is a 25fps scroll at 8-pixel jumps.
- **Using the shadow screen** (see below).

Horizontal Pixel Scrolling

Horizontal scrolling is the bread and butter of side-scrolling games: the world moves left or right as the player walks. And it is the most expensive type of scrolling on the Spectrum, because it requires not just copying bytes but *shifting* them.

Why Horizontal Scrolling Is Expensive

When you scroll the screen left by one pixel, every byte in every row must shift its bits left by one position, and the bit that falls off the left edge of one byte must become the rightmost bit of its left neighbour. This is a rotate-and-carry chain across all 32 bytes of each row.

The Z80's `RL` (rotate left through carry) instruction is the tool for this. For a leftward scroll, each pixel moves one position left. Bit 7 is the leftmost pixel in a byte, bit 0 the rightmost. Shifting left means each byte's bit 7 exits and must enter bit 0 of the byte to its left. The carry flag bridges adjacent bytes, so we process the row from **right to left**:

"'z80 id:ch17_why_horizontal_scrolling_is ; Scroll one pixel row left by 1 pixel ; HL points to byte 31 (rightmost) of the row ; ; Process right to left. Each byte rotates left; carry propagates. ; or a ; 4 T clear carry (no pixel entering from right)

```
; Byte 31 (rightmost)
rl   (hl)                ; 15 T  shift left, bit 7 -> carry, carry -> bit 0
dec  hl                  ; 6 T
; Byte 30
rl   (hl)                ; 15 T
dec  hl                  ; 6 T
; ...repeat for bytes 29 down to 0...
; Byte 0 (leftmost)
rl   (hl)                ; 15 T  bit 7 of byte 0 is lost (scrolled off screen)


Each byte costs: 15 (RL (HL)) + 6 (DEC HL) = **21 T-states** per byte. For 32 bytes per row: 3
states** per row (we do not need the final DEC HL).

Actually, the first byte needs `OR A` (4 T) to clear carry. So one row costs: 4 + 32 x 15 + 31
states**.

For 192 rows: 192 x 670 = **128,640 T-states**. That is **179% of a frame**.

A full-screen horizontal pixel scroll by one pixel does not fit in a single frame using RL cha

### The Full Budget Calculation

Let us lay out the complete per-row cost with all the overhead of navigating the interleaved s

| Operation | T-states per row |
|-----------|-----------------|
| Set HL to row start (or advance from previous) | ~15 |
| Set HL to rightmost byte: `ld a, l : or $1F : ld l, a` | 15 |
| Clear carry: `or a` | 4 |
```

32 x `rl (hl)`	480
31 x `dec hl` (between bytes)	186
Advance to next row (`inc h` or boundary cross)	4--77
Total per row (typical)	**~704**

For 192 rows: 192 x 704 = **135,168 T-states** = **189% of one frame**.

For a 168-row play area: 168 x 704 = **118,272 T-states** = **165% of one frame**.

There is no way to do a full-screen single-pixel horizontal scroll in one frame with standard

Can We Do Better?

You might think unrolling or alternate addressing modes would help. They do not. `RL (IX+d)` c
states -- *more* than `RL (HL)` at 15 T. A load-rotate-store sequence (`LD A,(HL) : RLA : LD (

Bottom line: The only way to make horizontal scrolling affordable is to reduce the number

---

Attribute (Character) Scrolling

If pixel scrolling is expensive, attribute scrolling is almost free by comparison. Attribute s
pixel (one character cell) jumps. You only move the 768 bytes of attribute memory and the corr
aligned pixel blocks -- or, more commonly, you move just the attributes and redraw the play ar

Scrolling Attributes with LDIR

The attribute area is linear: 32 bytes per row, 24 rows, sequential from `$5800` to `$5AFF`. S
31 to positions 0--30 in each row, then writing the new column at position 31.

For the entire 24-row attribute area:

```z80 id:ch17_scrolling_attributes_with
; Scroll all attributes left by 1 character column
; New column data in a 24-byte table at new_col_data
;
scroll_attrs_left:
    ld   hl, $5801          ; 10 T  source: column 1
    ld   de, $5800          ; 10 T  dest: column 0
    ld   bc, 767            ; 10 T  768 - 1 bytes
    ldir                    ; 767*21 + 16 = 16,123 T

    ; Now fill the rightmost column with new data
    ld   hl, new_col_data   ; 10 T
    ld   de, $581F          ; 10 T  column 31 of row 0
    ld   b, 24              ; 7 T
.fill_col:
    ld   a, (hl)            ; 7 T
    ld   (de), a            ; 7 T
    inc  hl                 ; 6 T
```

```
    ; advance DE by 32 (next attribute row)
    ld   a, e               ; 4 T
    add  a, 32              ; 7 T
    ld   e, a               ; 4 T
    jr   nc, .no_carry      ; 12/7 T
    inc  d                  ; 4 T
.no_carry:
    djnz .fill_col          ; 13 T
    ret

    ; Total LDIR: ~16,123 T
    ; Total column fill: ~24 * 50 = ~1,200 T
    ; Grand total: ~17,323 T = 24.2% of frame
```

17,323 T-states for a full-screen attribute scroll. That is about 24% of a frame. Compare this to the 135,000+ T-states for a pixel scroll. Attribute scrolling is nearly 8x cheaper.

The catch: the scroll jumps by 8 pixels at a time. The visual result is coarse and jerky. For text scrollers in demos, this is often acceptable – the viewer reads the text, not the smoothness. For a game, 8-pixel jumps feel terrible. That is where the combined method comes in.

The Combined Method: Character Scroll + Pixel Offset

This is the technique that most Spectrum side-scrolling games actually use. The idea is simple and powerful:

1. Maintain a **pixel offset** counter from 0 to 7. Each frame, increment the offset.
2. When the offset reaches 8, reset it to 0 and perform an **attribute/character scroll** – the cheap operation.
3. On every frame, render the play area with the current pixel offset applied. This offset shifts the entire display by 0–7 pixels within the current character column positions.

The pixel offset can be applied in two ways:

Method A: Shift the new column. Only shift the one column of pixel data (the column being scrolled into view) by the current offset. The rest of the screen is drawn from tiles at character alignment. This works when you have a tile-based renderer that redraws from a map.

Method B: Hardware-style virtual offset. Maintain a rendering offset that controls where within each character cell the tile data begins. This is conceptually similar to a hardware scroll register but implemented in software.

Method A is more common in practice. Let us walk through it.

How It Works

Imagine the play area is 20 characters (160 pixels) wide and 20 characters tall. The level data is a tilemap where each tile is 8x8 pixels (one character cell).

The scroll state consists of: - `scroll_tile_x`: which tile column is at the left edge of the screen (integer, advances by 1 every 8 frames). - `scroll_pixel_x`: pixel offset within the current tile (0–7, advances by 1 each frame).

Each frame:

1. **If `scroll_pixel_x` is 0:** Redraw the entire play area from the tilemap at character alignment. This is a tile renderer, which we can make fast using LDIR or LDI chains (each tile row is 1 byte or a few bytes of data copied to the right screen address). Cost: 20 columns x 20 rows x ~100 T per tile = ~40,000 T. Affordable.
2. **If `scroll_pixel_x` is 1–7:** Redraw the play area shifted by `scroll_pixel_x` pixels. For most of the play area, the tiles are character-aligned and can be drawn normally – the pixel offset only affects the **leftmost and rightmost visible columns**, where a tile is partially visible.

Wait – that is the efficient interpretation, but it requires a tile renderer that clips at sub-character boundaries. The simpler (and more common) approach is:

The Simple Combined Method

1. Every 8 frames, perform a character-level scroll (LDIR the attribute and pixel data left by one column). Cost: ~17,000 T for attributes + ~40,000 T for pixel data = ~57,000 T. Done once every 8 frames.
2. Every frame, shift a **narrow window** by 1 pixel. This window is only 1 column (32 bytes) or 2 columns (64 bytes) wide – the seam between the old data and the newly arriving column.
3. **Between character scrolls**, the display shows the last character-scrolled position with a 0–7 pixel offset applied to the edge column. The player perceives smooth 1-pixel-per-frame scrolling.

Here is the per-frame cost breakdown:

Operation	T-states	Frequency
Character scroll (full play area)	~57,000	Every 8th frame
Pixel-shift 1–2 edge columns (20 rows x 2 cols x 21 T/byte x 8 scanlines)	~6,720	Every frame
Draw new tile column at right edge	~5,000	Every 8th frame
Attribute column update	~1,200	Every 8th frame

On 7 out of 8 frames: ~6,720 T-states for the edge pixel shift. That is under 10% of the frame budget. Plenty of room for game logic, sprites, and music.

On every 8th frame: ~6,720 + 57,000 + 5,000 + 1,200 = ~69,920 T-states. That is 97.5% of the frame budget. Tight, but doable – especially if you split the character scroll across two frames or use the shadow screen.

Implementation: The Edge-Column Pixel Shift

The key inner routine shifts 1 or 2 columns of pixel data by 1 pixel. For a 2-column (16-pixel) window, each row has 2 bytes to shift:

```
z80 id:ch17_implementation_the_edge ; Shift 2 bytes left by 1 pixel with carry
propagation ; HL points to the right byte of the pair ;     or   a                ;
4 T    clear carry     rl   (hl)             ; 15 T   right byte: shift left,
bit 7 -> carry     dec  hl               ; 6 T     rl   (hl)             ; 15 T
left byte: carry -> bit 0, bit 7 lost                          ; total: 40 T
per row (for 2-byte window)
```

For 160 rows (20 char rows x 8 scanlines): 160 x 40 = **6,400 T-states**. With pointer advancement overhead (~20 T per row), the total is about **9,600 T-states** per frame. Very affordable.

The Rendering Pipeline

Here is the complete per-frame sequence for a combined horizontal scroller:

“‘z80 id:ch17_the_rendering_pipeline frame_loop: halt ; wait for interrupt

```
; --- Always: advance pixel offset ---
ld   a, (scroll_pixel_x)
inc  a
cp   8
jr   nz, .no_char_scroll

; --- Every 8th frame: character scroll ---
xor  a                       ; reset pixel offset to 0
ld   (scroll_pixel_x), a

; Advance tile position
ld   hl, (scroll_tile_x)
inc  hl
ld   (scroll_tile_x), hl

; Scroll pixel data left by 1 column (8 pixels)
call scroll_pixels_left_char

; Scroll attributes left by 1 column
call scroll_attrs_left

; Draw new tile column on right edge
call draw_right_column

jr   .scroll_done
```

.no_char_scroll: ld (scroll_pixel_x), a

```
; Shift the edge columns by 1 pixel
call shift_edge_columns
```

.scroll_done: ; — Game logic, sprites, music — call update_entities call draw_sprites call play_music

```
jr   frame_loop


This is the skeleton of a real Spectrum side-scroller. The key insight is that smooth 1-
pixel scrolling is achieved *without* shifting the entire screen every frame. The expensive ch
level scroll happens only once every 8 frames, and the per-frame work is minimal.

---

## Scrolling the Pixel Data by One Character Column

The character-level pixel scroll (step 2 in the pipeline above) shifts 8 pixels' worth of data
level* copy, not a bit-level rotate. Each row's 32 bytes shift left by 1 byte: byte[1] goes to

For a single row, this is a 31-byte LDIR:

```z80 id:ch17_scrolling_the_pixel_data_by
; Shift one pixel row left by 8 pixels (1 byte)
; HL = address of byte 1 (source), DE = address of byte 0 (dest)
; BC = 31
;
    ldir                    ; 31*21 - 5 = 646 T per row... wait.
                            ; Actually: 30*21 + 16 = 646 T. Yes.
```

For the full play area (168 rows): 168 x 646 = 108,528 T-states + row navigation overhead.

A better approach leverages the fact that within each scanline of a character row, the bytes are contiguous. For 20 character columns, one scanline's data is 20 contiguous bytes. Scrolling that scanline left by 1 byte means LDIR of 19 bytes:

```
z80 id:ch17_scrolling_the_pixel_data_by_2 ; Scroll one scan line of the play
area left by 1 character column ; Play area is 20 columns wide (columns 2-
21, for example) ; Source: column 3, Dest: column 2, count: 19 ;     ld   hl,
row_addr + 3    ; source = byte 3 of this scan line     ld   de, row_addr +
2    ; dest   = byte 2     ld   bc, 19              ; 19 bytes to copy     ldir
; 18*21 + 16 = 394 T
```

For 160 rows: 160 x 394 = 63,040 T-states. Add ~20 T per row for pointer navigation: 160 x 414 = **66,240 T-states**. That is 92% of a frame. Doable but tight on the "every 8th frame" budget.

With LDI chains (19 LDIs per row): 19 x 16 = 304 T per row. For 160 rows: 160 x 324 = **51,840 T-states** = 72% of a frame. Now we have 28% left for drawing the new column and updating attributes.

The Shadow Screen Trick

The ZX Spectrum 128K has a feature that transforms the scrolling problem: **two screen buffers**. The standard screen lives at `$4000` in page 5 (always mapped at `$4000–$7FFF`). The shadow screen lives at `$C000` in page 7 (mapped at `$C000–$FFFF` when page 7 is banked in).
```

Port `$7FFD` controls which screen is displayed:

“‘z80 id:ch17_the_shadow_screen_trick ; Bit 3 of port $7FFD selects the display screen: ; Bit 3 = 0: display page 5 (standard screen at $4000) ; Bit 3 = 1: display page 7 (shadow screen at $C000)

```
ld   a, (current_bank)
or   %00001000           ; set bit 3: display shadow screen
ld   bc, $7FFD
out  (c), a
```

```
The trick for scrolling:

1. **Frame N:** The player sees the standard screen (page 5). Meanwhile, you draw the *next* f
2. **Frame N+1:** Flip the display to the shadow screen. The player now sees the freshly drawn
hidden standard screen.

This double-buffering approach eliminates tearing completely and gives you a full frame (or mo
```

```z80 id:ch17_the_shadow_screen_trick_2
; Flip displayed screen and return back buffer address in HL
;
; screen_flag:  0 = showing page 5, drawing to page 7
;               1 = showing page 7, drawing to page 5
;
flip_screens:
    ld   a, (screen_flag)
    xor  1                   ; 7 T   toggle (XOR with immediate)
    ld   (screen_flag), a

    ld   hl, $C000           ; assume drawing to page 7
    or   a
    jr   z, .show_page5

    ; Now showing page 7, draw to page 5
    ld   hl, $4000
    ld   a, (current_bank)
    or   %00001000           ; bit 3 set: display page 7
    jr   .do_flip

.show_page5:
    ld   a, (current_bank)
    and  %11110111           ; bit 3 clear: display page 5

.do_flip:
    ld   bc, $7FFD
    out  (c), a
    ld   (current_bank), a
    ret                      ; HL = back buffer address
```

Shadow Screen Scrolling Strategy

With double-buffering, the scrolling approach changes:

Instead of scrolling the live screen in place (which causes tearing and must complete within one frame), you **redraw the play area from the tilemap** into the back buffer at the new scroll position. This is fundamentally different. You are not *moving* existing screen data – you are *rendering fresh* from the map.

This is more work per frame (you redraw the whole play area, not just shift it), but it has significant advantages:

1. **No tearing.** The player never sees a half-scrolled screen.
2. **No edge-column shifting.** You render each tile at its correct sub-character offset directly.
3. **Flexible scroll speed.** You can scroll 1, 2, or 3 pixels per frame without changing the rendering logic.
4. **Simpler code.** A tile renderer is simpler than a combined shift-and-copy scroller.

The cost of a full play-area redraw from tiles depends on your tile renderer. With 20 x 20 tiles, each tile being 8 bytes (8 scanlines x 1 byte), and using LDI chains:

- 400 tiles x 8 bytes x 16 T per LDI = 51,200 T-states for data output.
- Plus tile address lookups and screen address calculations: ~20 T per tile x 400 = 8,000 T.
- **Total: ~59,200 T-states** = 82% of a frame.

This leaves 18% (~12,900 T-states) for sprites, game logic, and music. Tight but workable.

Comparison: Scrolling Methods on ZX Spectrum

Method	T-states/frame	% of frame	Visual quality	Notes
Full pixel scroll (horizontal, 1px)	~135,000	189%	Smooth	Impossible at 50fps
Full pixel scroll (vertical, 1px)	~107,000	149%	Smooth	Impossible at 50fps
Attribute scroll only	~17,000	24%	Jerky (8px jumps)	Very cheap
Combined (char + pixel edge)	~10,000 avg, ~70,000 peak	14%/98%	Smooth	Best single-buffer method
Shadow screen + tile redraw	~59,000	82%	Smooth, tear-free	Requires 128K

Method	T-states/frame	% of frame	Visual quality	Notes
Character scroll (8px jumps)	~52,000–66,000	73–92%	Jerky	For scrolling text/status

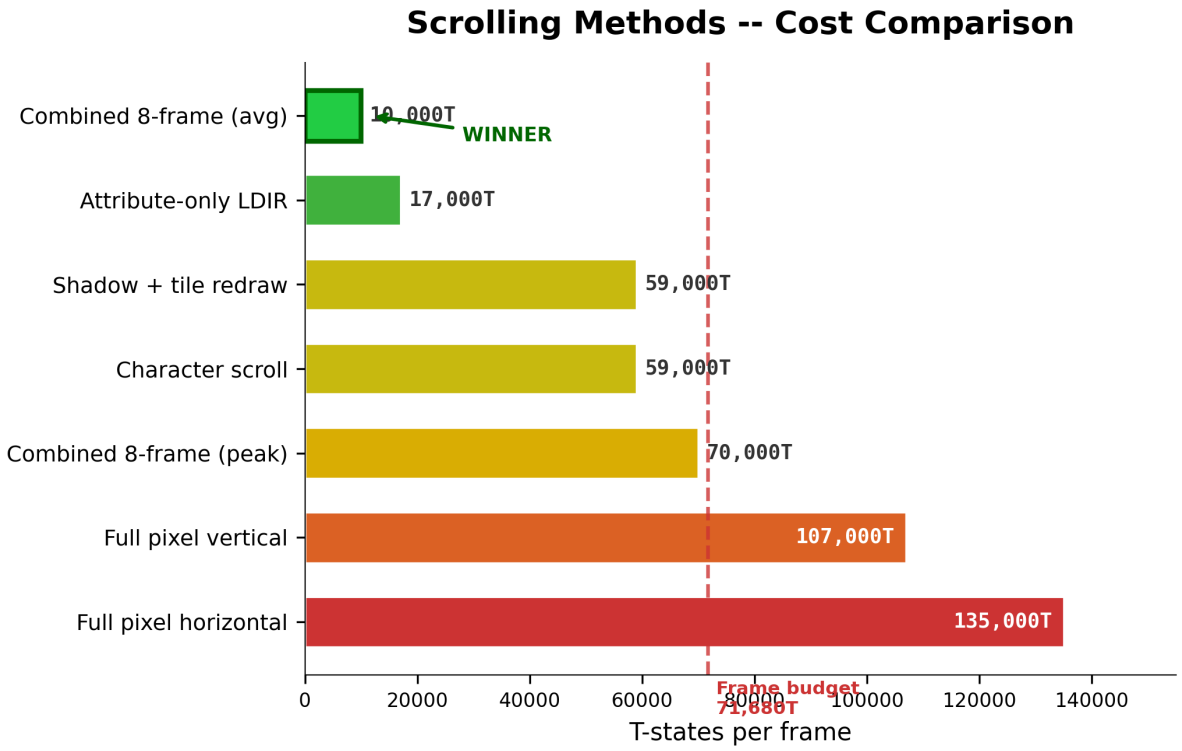


Figure 12: Scrolling technique cost comparison

Scrolling Right (and the Direction Problem)

Everything above describes a leftward scroll (the player moves right, the world shifts left). What about scrolling right?

For attribute scrolling, reverse the LDIR direction. Copy bytes 0–30 to positions 1–31, right to left. LDIR copies forward (low to high addresses), so for a rightward scroll you need LDDR (copy backward):

```
z80 id:ch17_scrolling_right_and_the ; Scroll attributes right by 1 character
column ; scroll_attrs_right:     ld   hl, $5ADE          ; source: last row,
column 30     ld   de, $5ADF          ; dest: last row, column 31     ld   bc,
767            ; 768 - 1 bytes     lddr                    ; 767*21 + 16 =
16,123 T     ret
```

For pixel bit-shifting, a rightward scroll uses `RR (HL)` instead of `RL (HL)`, processing left to right:

```
z80 id:ch17_scrolling_right_and_the_2 ; Scroll one pixel row RIGHT by 1 pixel
; HL points to byte 0 (leftmost) ;     or   a               ; 4 T    clear
carry     rr   (hl)            ; 15 T   shift right, bit 0 -> carry     inc
hl               ; 6 T     rr   (hl)             ; 15 T   carry -> bit 7     inc
hl               ; 6 T     ; ... 32 bytes total ...
```

The per-byte cost is identical: 21 T-states. A rightward scroll costs the same as a leftward scroll. The combined method works in both directions with the same budget.

For bidirectional scrolling (the player can go left or right), you need two versions of the character-scroll and edge-shift routines, switched based on direction. Self-modifying code is useful here: before the scroll, patch the RL/RR opcode and the INC/DEC direction in the shift routine. This avoids a branch inside the inner loop (see Chapter 3 for the SMC pattern).

Agon Light 2: Hardware Scrolling

The Agon Light 2's VDP (Video Display Processor) handles scrolling entirely differently from the Spectrum. Where the Spectrum programmer must move bytes manually, the Agon provides hardware-level support for scroll offsets and tilemaps.

Hardware Scroll Offsets

The VDP supports a viewport offset for bitmap modes. By setting the scroll offset registers, you shift the entire displayed image without moving any pixel data. The eZ80 sends a VDP command via the serial link:

```
z80 id:ch17_hardware_scroll_offsets ; Agon: set horizontal scroll offset ; VDU
23, 0, &C3, x_low, x_high ;     ld   a, 23     call vdu_write      ; VDU command
prefix     ld   a, 0     call vdu_write     ld   a, $C3         ; set scroll
offset command     call vdu_write     ld   a, (scroll_x)     call vdu_write
; x offset low byte     ld   a, (scroll_x+1)     call vdu_write      ; x offset
high byte
```

The hardware applies this offset when reading the framebuffer for display. No pixel data is moved, no CPU T-states are spent on shifting bytes, and the scroll is perfectly smooth at any speed. The CPU cost is just the serial communication overhead (a few hundred T-states for the VDU command sequence).

Tilemap Scrolling

The VDP's tilemap mode provides native tile-based rendering. You define a set of tiles (8x8 or 16x16 pixel patterns), build a map array that references tile indices, and the hardware renders the map at display time. Scrolling is achieved by changing the tilemap's viewport offset:

```
z80 id:ch17_tilemap_scrolling ; Agon: set tilemap scroll offset ; VDU 23, 27,
<tilemap_scroll_command>, offset_x, offset_y ;     ld   a, 23     call vdu_write
ld   a, 27     call vdu_write     ld   a, 14          ; set tilemap scroll
offset     call vdu_write     ; ... send x and y offsets ...
```

The tilemap wraps around automatically. As the viewport scrolls past the edge of the map, the hardware wraps to the beginning (or you can update the edge column with new tile indices – the ring-buffer column loading technique).

Ring-Buffer Column Loading

For an infinitely scrolling level, the tilemap acts as a ring buffer. The map is wider than the screen by at least one column. As the player scrolls right:

1. The hardware scroll offset advances by 1 pixel per frame (or whatever speed you want).
2. When a new tile column is about to scroll into view, the eZ80 writes new tile indices into the column that just scrolled off the left edge.
3. The tilemap wraps, and the newly written column appears on the right.

“‘z80 id:ch17_ring_buffer_column_loading ; Ring-buffer column loading (Agon, conceptual) ; ; tilemap is 40 columns wide, screen shows 32 ; scroll_col tracks which column is at the left edge ; ring_buffer_load: ld a, (scroll_col) add a, 32 ; column about to appear on right and 39 ; wrap to tilemap width (mod 40) ld c, a ; C = column index to update

```
; Load new tile data for this column from the level map
; (level_map is a wider array of tile indices)
ld   hl, (level_ptr)     ; pointer into the level data
ld   b, 20               ; 20 rows
```

.load_col: ld a, (hl) ; read tile index from level inc hl ; Write tile index to tilemap at (C, row) call set_tilemap_cell ; VDP command to set one cell djnz .load_col

```
ld   (level_ptr), hl
ret


The CPU work per frame is minimal: writing 20 tile indices via VDP commands, perhaps 2,000--
3,000 T-states total. The rest of the frame is available for game logic. Compare this to the S
states for a tile-based scroll redraw. The Agon's hardware tilemap buys you roughly a 20x redu

### Comparison: Spectrum vs. Agon Scrolling

| Aspect | ZX Spectrum | Agon Light 2 |
|--------|-------------|---------------|
| Scroll granularity | Software-limited; 1px possible but expensive | 1px native, zero CPU cos
| CPU cost per frame | 10,000--135,000 T | 500--3,000 T |
| Tearing | Visible without double-buffering | None (VDP handles sync) |
| Direction change | Requires alternate routines or SMC | Change offset sign |
| Map size limit | Limited by RAM, no hardware support | Tilemap size limited by VDP memory |
| Colour per tile | 2 colours per 8x8 cell (attribute) | Full colour per pixel |

The contrast is stark. What the Spectrum programmer spends most of their frame budget on -- mo
screen tricks. The Agon's constraints are elsewhere (serial VDP latency, command overhead for

---

## Practical: Horizontal Side-Scrolling Level
```

```
### Spectrum Version: Combined Character + Pixel Scroll

Build a horizontal side-scroller with a 20x20-character play area that scrolls smoothly at 1 p

Here is the complete structure:

```z80 id:ch17_spectrum_version_combined
; Side-scroller engine — ZX Spectrum 128K
; Uses combined character + pixel method with shadow screen.
;
    ORG $8000

PLAY_X      EQU 2           ; play area starts at column 2
PLAY_Y      EQU 2           ; play area starts at char row 2
PLAY_W      EQU 20          ; play area width in characters
PLAY_H      EQU 20          ; play area height in characters

scroll_pixel_x:   DB 0      ; pixel offset 0-7
scroll_tile_x:    DW 0      ; tile column at left edge
screen_flag:      DB 0      ; which screen is visible
current_bank:     DB 0      ; current $7FFD value

; --- Main loop ---
main:
    halt                    ; 4 T   sync to frame

    ; Advance scroll
    ld   a, (scroll_pixel_x) ; 13 T
    inc  a                  ; 4 T
    cp   8                  ; 7 T
    jr   c, .pixel_only     ; 12/7 T

    ; Character scroll frame
    xor  a
    ld   (scroll_pixel_x), a

    ; Advance tile position
    ld   hl, (scroll_tile_x)
    inc  hl
    ld   (scroll_tile_x), hl

    ; Get back buffer address
    call get_back_buffer    ; HL = $4000 or $C000

    ; Redraw full play area from tilemap into back buffer
    call render_play_area   ; ~50,000 T

    ; Flip screens
    call flip_screens       ; ~30 T
```
```

```
    jr   .frame_done

.pixel_only:
    ld   (scroll_pixel_x), a

    ; Shift edge columns in current (non-displayed) buffer
    call get_back_buffer
    call shift_edge_columns  ; ~9,600 T

    call flip_screens

.frame_done:
    call update_player       ; ~2,000 T
    call draw_sprites        ; ~5,000 T
    call play_music          ; ~3,000 T (IM2 handler)

    jr   main

; --- Render full play area from tilemap ---
; Input: HL = base address of target screen ($4000 or $C000)
;
render_play_area:
    ; For each tile in the play area:
    ;   Look up tile index from tilemap
    ;   Copy 8 bytes of tile data to screen, navigating interleave
    ;
    ; 20 columns x 20 rows = 400 tiles
    ; Each tile: 8 scan lines x 1 byte = 8 LDI operations
    ; Per tile: lookup (20 T) + 8 x (LDI 16 T + INC H 4 T) = 180 T
    ; Total: 400 x 180 = 72,000 T
    ;
    ; (Actual implementation uses PUSH tricks and
    ;  pre-computed screen address tables for ~55,000 T)
    ret

; --- Shift edge columns by 1 pixel ---
; Shifts the 2 rightmost columns of the play area left by 1 pixel
;
shift_edge_columns:
    ; For each of 160 pixel rows in the play area:
    ;   Navigate to the correct screen address
    ;   RL (HL) on the 2 edge bytes, right to left
    ;
    ; Per row: 40 T (2 bytes shifted) + 20 T (navigation)
    ; Total: 160 x 60 = 9,600 T
    ret
```

Agon Version: Hardware Tilemap Scrolling

The Agon version is dramatically simpler. The main loop calls `vsync`, increments a 16-bit scroll offset, sends it to the VDP via the `set_scroll_offset` routine (a handful

of `vdu_write` calls), and every 8 pixels calls `ring_buffer_load` to update one column of tile indices. The entire scroll costs under 3,000 T-states per frame, leaving 365,000+ T-states for game logic, AI, physics, and rendering. The Spectrum version is a careful exercise in cycle-counting where every technique from Chapters 2 and 3 comes together to achieve what the Agon does with a hardware register.

Vertical + Horizontal: Combined Scrolling

Some games scroll in both directions simultaneously. On the Spectrum, apply the combined method to both axes: character scroll + pixel offset (0–7) for each. The character scroll in each direction happens once every 8 frames. Both coinciding on the same frame is a 1/64 probability (about every 1.3 seconds) – either accept one dropped frame or split the work. The per-frame edge-shifting cost for both axes: horizontal edge columns (~9,600 T) + vertical edge rows (~6,400 T) = ~16,000 T = 22% of the frame. Manageable.

Optimisation Tips

1. Use a Screen Address Lookup Table

Pre-compute a table of 192 screen addresses (one per pixel row) in RAM. Cost: 384 bytes. Benefit: a 16-bit table lookup (about 30 T-states) replaces the bit-shuffling address calculation (91 T-states).

2. Scroll Only What Is Visible

If sprites cover part of the play area, you can skip scrolling the rows behind opaque sprites. Track which rows need scrolling with a dirty-row bitmap. This optimisation pays off when sprites cover a significant fraction of the play area.

3. Use PUSH for the Character Scroll

For the character-level pixel data scroll (copying 19 bytes left per scanline), the PUSH trick works well. Set SP to the end of each scanline's play area, POP 10 bytes, shift the register contents, and PUSH them back one byte offset. This is complex to set up but reduces the per-scanline cost by 30–40%.

4. Split the Character Scroll Across Frames

If the character scroll (every 8th frame) is too expensive for one frame, split it: scroll the top half of the play area on frame N and the bottom half on frame N+1. The visual artefact (the top half shifts 1 frame before the bottom) is barely noticeable at 50fps.

5. Palette and Attribute Tricks

For attribute-only scrolling (no pixel data involved), consider using FLASH or BRIGHT changes to create the illusion of motion within a static pixel grid. A rotating set of attribute colours on character-aligned tiles can simulate flow, water, or conveyor belts without moving any pixel data at all.

Summary

- **Full-screen pixel scrolling on the ZX Spectrum is impossible at 50fps.** Horizontal pixel scrolling costs ~135,000 T-states for 192 rows (189% of the frame budget). Vertical costs ~107,000 T-states (149%). The interleaved memory layout adds complexity to vertical scrolling, and the lack of a barrel shifter makes horizontal scrolling inherently expensive.
- **Attribute scrolling is cheap** at ~17,000 T-states (24% of frame), but moves in coarse 8-pixel jumps.
- **The combined method** is what real games use: character-level scrolls (every 8 frames) plus per-frame pixel-shifting of the 1–2 edge columns. Average per-frame cost is under 10,000 T-states. The spike on character-scroll frames (~70,000 T) can be managed with the shadow screen or by splitting across frames.
- **The shadow screen** (128K, page 7) provides tear-free double buffering. Draw the next frame into the back buffer, then flip the display. This changes the scrolling strategy from "shift existing data" to "redraw from tilemap," which is conceptually simpler and eliminates tearing.
- **Horizontal scrolling direction** does not change the cost. Rightward scrolling uses `RR (HL)` instead of `RL (HL)`, left-to-right instead of right-to-left, at the same 21 T-states per byte.
- **Vertical pixel scrolling** is complicated by the Spectrum's interleaved screen layout. Moving one pixel row down means navigating the `010TTSSS LLLCCCCC` address structure, with different pointer adjustments at character-cell and third boundaries. The split-counter approach from Chapter 2 is essential.
- **The Agon Light 2** provides hardware scroll offsets and tilemap rendering that reduce the CPU cost of scrolling to a few VDP commands per frame (~500–3,000 T-states). Ring-buffer column loading keeps the tilemap updated as new terrain scrolls into view. What the Spectrum programmer builds in 70,000 T-states, the Agon handles with a register write.
- **Key techniques from earlier chapters** are essential here: the interleaved screen layout and split-counter navigation (Chapter 2), LDI chains and PUSH tricks for fast data movement (Chapter 3), and self-modifying code for direction-switchable scroll routines (Chapter 3).

Sources: Introspec "Eshchyo raz pro DOWN_HL" (Hype, 2020) for interleaved screen navigation; Introspec "GO WEST Part 1" (Hype, 2015) for contended memory costs; DenisGrachev "Ringo Render 64x48" (Hype,

2022) for half-character displacement scrolling; ZX Spectrum 128K Technical Manual for port `$7FFD` and shadow screen; Agon Light 2 VDP documentation for tilemap and scroll offset commands.

Chapter 18: Game Loop and Entity System

> "A game is a demo that listens."

Every demo effect we have built so far runs in a closed loop: calculate, render, repeat. The viewer watches. The code does not care whether anyone is in the room. A game breaks this contract. A game *responds*. The player presses a key and something must change – immediately, reliably, within the same frame budget we have been counting since Chapter 1.

This chapter is about building the architecture that makes a game possible on the ZX Spectrum and Agon Light 2. Not the rendering (Chapter 16 covered sprites, Chapter 17 covered scrolling) and not the physics (Chapter 19 will handle collisions and AI). This chapter is the skeleton: the main loop that drives everything, the state machine that organises the flow from title screen to gameplay to game over, the input system that reads the player's intentions, and the entity system that manages every object in the game world.

By the end, you will have a working game skeleton with 16 active entities – a player, eight enemies, and seven bullets – running within the frame budget on both platforms.

18.1 The Main Loop

Every game on the ZX Spectrum follows the same fundamental rhythm:

```
1. HALT          -- wait for the frame interrupt
2. Read input    -- what does the player want?
3. Update state  -- move entities, run AI, check collisions
4. Render        -- draw the frame
5. Go to 1
```

This is the game loop. It is not complicated. Its power comes from the fact that it runs fifty times per second, every second, and everything the player experiences emerges from this cycle.

Here is the minimal implementation:

"'z80 id:ch18_the_main_loop_2 ORG $8000

```
; Install IM1 interrupt handler (standard for games)
im   1
ei
```

main_loop: halt ; 4T + wait – sync to frame interrupt

```
call read_input         ; poll keyboard/joystick
call update_entities    ; move everything, run logic
call render_frame       ; draw to screen

jr   main_loop          ; 12T -- loop forever
```

```
The `HALT` instruction is the heartbeat. When the CPU executes `HALT`, it stops and waits for

This gives you exactly one frame's worth of T-states to do everything. If your work finishes e

### The Frame Budget, Revisited

We established the numbers in Chapter 1, but they bear repeating in the context of a game:

| Machine | T-states per frame | Practical budget |
|---------|--------------------|------------------|
| ZX Spectrum 48K | 69,888 | ~62,000 (after interrupt overhead) |
| ZX Spectrum 128K | 70,908 | ~63,000 |
| Pentagon 128 | 71,680 | ~64,000 |
| Agon Light 2 | ~368,640 | ~360,000 |

The "practical budget" accounts for the interrupt handler, the `HALT` instruction itself, and
states of usable time per frame. On the Agon, you have over five times that.

How does a typical game spend those 64,000 T-states? Here is a realistic breakdown for a Spect

| Subsystem | T-states | % of budget |
|-----------|----------|-------------|
| Input reading | ~500 | 0.8% |
| Entity update (16 entities) | ~8,000 | 12.5% |
| Collision detection | ~4,000 | 6.3% |
| Music player (PT3) | ~5,000 | 7.8% |
| Sprite rendering (8 visible) | ~24,000 | 37.5% |
| Background/scroll update | ~12,000 | 18.8% |
| Miscellaneous (HUD, state) | ~3,000 | 4.7% |
| **Remaining headroom** | **~7,500** | **11.7%** |

That 11.7% headroom is your safety margin. Eat into it and you start dropping frames on comple
colour profiling technique from Chapter 1 -- red for sprites, blue for music, green for logic

On the Agon, the same game logic runs in a fraction of the budget. The entity update, collisio
states total -- about 4% of the Agon's frame. The VDP handles sprite rendering on the ESP32 co
side sprite cost drops to the VDU command overhead. You have enormous room for more complex AI

<!-- figure: ch18_game_loop -->
![Game loop architecture](illustrations/output/ch18_game_loop.png)
```

```
---

## 18.2 The Game State Machine

A game is not one loop -- it is several. The title screen has its own loop (animate logo, wait
over screen has yet another.

The cleanest way to organise these is a **state machine**: a variable that tracks which state

### State Definitions

```z80 id:ch18_state_definitions
; Game states (byte values, used as table offsets)
STATE_TITLE     EQU  0
STATE_MENU      EQU  2      ; x2 because each table entry is 2 bytes
STATE_GAME      EQU  4
STATE_PAUSE     EQU  6
STATE_GAMEOVER  EQU  8

; Current state variable
game_state:     DB   STATE_TITLE
```

The Jump Table

```
z80 id:ch18_the_jump_table ; Table of handler addresses, indexed by state state_table:
DW   state_title        ; STATE_TITLE   = 0     DW   state_menu         ; STATE_MENU
= 2     DW   state_game         ; STATE_GAME    = 4     DW   state_pause        ;
STATE_PAUSE   = 6     DW   state_gameover     ; STATE_GAMEOVER = 8
```

The Dispatcher

The main loop becomes a dispatcher that reads the current state and jumps to the appropriate handler:

"'z80 id:ch18_the_dispatcher main_loop: halt ; sync to frame

```
; Dispatch to current state handler
ld   a, (game_state)    ; 13T  load state index
ld   l, a               ; 4T
ld   h, 0               ; 7T
ld   de, state_table    ; 10T
add  hl, de             ; 11T  HL = state_table + offset
ld   e, (hl)            ; 7T   low byte of handler address
inc  hl                 ; 6T
ld   d, (hl)            ; 7T   high byte of handler address
ex   de, hl             ; 4T   HL = handler address
jp   (hl)               ; 4T   jump to handler
                        ; --- 73T total dispatch overhead
```

```
The `JP (HL)` instruction is the key. It does not jump to the address stored *at* HL -- it jum
states. The entire dispatch -- loading the state variable, computing the table offset, reading
```
```

states. That is negligible: about 0.1% of the frame budget.

Each handler runs its own logic and then jumps back to `main_loop`:

```z80 id:ch18_the_dispatcher_2
state_title:
    call draw_title_screen
    call read_input
    ; Check for start key (SPACE or ENTER)
    ld   a, (input_flags)
    bit  BUTTON_FIRE, a
    jr   z, .no_start
    ; Transition to menu
    ld   a, STATE_MENU
    ld   (game_state), a
    call init_menu          ; set up menu screen
.no_start:
    jp   main_loop

state_game:
    call read_input
    ; Check for pause
    ld   a, (input_keys)
    bit  KEY_P, a
    jr   z, .not_paused
    ld   a, STATE_PAUSE
    ld   (game_state), a
    jp   main_loop
.not_paused:
    call update_entities
    call check_collisions
    call render_frame
    call update_music       ; AY player -- see Chapter 11
    jp   main_loop

state_pause:
    ; Game is frozen -- only check for unpause
    call read_input
    ld   a, (input_keys)
    bit  KEY_P, a
    jr   z, .still_paused
    ld   a, STATE_GAME
    ld   (game_state), a
.still_paused:
    ; Optionally blink "PAUSED" text
    call blink_pause_text
    jp   main_loop

state_gameover:
    call draw_gameover_screen
    call read_input
```

```
    ld   a, (input_flags)
    bit  BUTTON_FIRE, a
    jr   z, .wait
    ld   a, STATE_TITLE
    ld   (game_state), a
    call init_title
.wait:
    jp   main_loop
```

Why Not a Chain of Comparisons?

You might be tempted to write the dispatcher as:

```
z80 id:ch18_why_not_a_chain_of     ld   a, (game_state)     cp   STATE_TITLE
jp   z, state_title     cp   STATE_MENU     jp   z, state_menu     cp   STATE_GAME
jp   z, state_game     ; ...
```

This works, but it has two problems. First, the cost grows linearly: each additional state adds a `CP` (7T) and a `JP Z` (10T), so the worst case is 17T per state. With 5 states, the game state (the most common case) might take 51T to reach if it is the third comparison. The jump table takes 73T regardless of which state is active – it is O(1), not O(n).

Second, and more importantly, the jump table scales cleanly. Adding a sixth state (say, STATE_SHOP) means adding one `DW` entry to the table and one constant definition. The dispatcher code does not change at all. With comparison chains, you add more instructions to the dispatcher itself, and the ordering starts to matter for performance. The table approach is both faster in the common case and cleaner to maintain.

State Transitions

State transitions happen by writing a new value to `game_state`. Typically you also call an initialisation routine for the new state:

```
z80 id:ch18_state_transitions ; Transition: Game -> Game Over game_over_transition:
ld   a, STATE_GAMEOVER     ld   (game_state), a     call init_gameover       ;
set up game over screen, save score     ret
```

Keep transitions explicit and centralised. A common bug in Z80 games is a state transition that forgets to initialise the new state's data – the game-over screen shows garbage because nobody cleared the screen or reset the animation counter. Every state should have an `init_` routine that the transition calls.

18.3 Input: Reading the Player

ZX Spectrum Keyboard

The Spectrum keyboard is read through port `$FE`. The keyboard is wired as a matrix of 8 half-rows, each selected by setting a bit low in the high byte of the port address. Reading port `$FE` with a specific high byte returns the state of that half-row: 5 bits, one per key, where 0 means pressed and 1 means not pressed.

The half-row map:

High byte	Keys (bit 0 to bit 4)
$FE (bit 0 low)	SHIFT, Z, X, C, V
$FD (bit 1 low)	A, S, D, F, G
$FB (bit 2 low)	Q, W, E, R, T
$F7 (bit 3 low)	1, 2, 3, 4, 5
$EF (bit 4 low)	0, 9, 8, 7, 6
$DF (bit 5 low)	P, O, I, U, Y
$BF (bit 6 low)	ENTER, L, K, J, H
$7F (bit 7 low)	SPACE, SYMSHIFT, M, N, B

The standard game controls – Q/A/O/P for up/down/left/right and SPACE for fire – span three half-rows. Here is a routine that reads them and packs the result into a single byte:

“‘z80 id:ch18_zx_spectrum_keyboard ; Input flag bits INPUT_RIGHT EQU 0 INPUT_LEFT EQU 1 INPUT_DOWN EQU 2 INPUT_UP EQU 3 INPUT_FIRE EQU 4

; Read QAOP+SPACE into input_flags ; Returns: A = input_flags byte, also stored at (input_flags) read_keyboard: ld d, 0 ; 7T accumulate result in D

```
; Read O and P: half-row $DF (P=bit0, O=bit1)
ld   bc, $DFFE          ; 10T
in   a, (c)             ; 12T
bit  0, a               ; 8T   P key
jr   nz, .no_right      ; 12/7T
set  INPUT_RIGHT, d     ; 8T
```

.no_right: bit 1, a ; 8T O key jr nz, .no_left ; 12/7T set INPUT_LEFT, d ; 8T .no_left:

```
; Read Q and A: half-rows $FB (Q=bit0) and $FD (A=bit0... wait)
; Q is in half-row $FB at bit 0
ld   b, $FB             ; 7T
in   a, (c)             ; 12T
bit  0, a               ; 8T   Q key
jr   nz, .no_up         ; 12/7T
set  INPUT_UP, d        ; 8T
```

.no_up:

```
; A is in half-row $FD at bit 0
ld   b, $FD             ; 7T
in   a, (c)             ; 12T
bit  0, a               ; 8T   A key
jr   nz, .no_down       ; 12/7T
set  INPUT_DOWN, d      ; 8T
```

.no_down:

```
; SPACE: half-row $7F at bit 0
ld   b, $7F             ; 7T
in   a, (c)             ; 12T
```

```
bit  0, a              ; 8T
jr   nz, .no_fire      ; 12/7T
set  INPUT_FIRE, d     ; 8T
```

.no_fire:

```
ld   a, d              ; 4T
ld   (input_flags), a  ; 13T
ret                    ; 10T
; Total: ~220T worst case (all keys pressed)


At roughly 220 T-states worst case, input reading is trivial in the frame budget. Even on the

### Kempston Joystick

The Kempston interface is even simpler. One port read returns all five directions plus fire:

```z80 id:ch18_kempston_joystick
; Kempston joystick port
KEMPSTON_PORT  EQU  $1F

; Read Kempston joystick
; Returns: A = joystick state
;   bit 0 = right, bit 1 = left, bit 2 = down, bit 3 = up, bit 4 = fire
read_kempston:
    in   a, (KEMPSTON_PORT)  ; 11T
    and  %00011111           ; 7T   mask to 5 bits
    ld   (input_flags), a    ; 13T
    ret                      ; 10T
    ; Total: 41T
```

Notice something convenient: the Kempston bit layout matches our `INPUT_*` flag definitions exactly. This is not a coincidence – the Kempston interface was designed with this standard in mind, and most Spectrum games adopt the same bit ordering. If you support both keyboard and joystick, you can OR the results together:

```
z80 id:ch18_kempston_joystick_2 read_input:     call read_keyboard       ;
D = keyboard flags     push de     call read_kempston       ; A = joystick
flags     pop  de     or   d                   ; combine both sources     ld
(input_flags), a     ret
```

Now the rest of your code only checks `input_flags` and does not care whether the input came from the keyboard or a joystick.

Edge Detection: Press vs Hold

For some actions – firing a bullet, opening a menu – you want to respond to the *press* event, not the held state. If you check `bit INPUT_FIRE, a` every frame, the player fires a bullet every 1/50th of a second while holding the button. That might be intentional for rapid-fire, but for a single-shot weapon or a menu selection, you need edge detection.

The technique: store the previous frame's input alongside the current frame's, and XOR them to find the bits that changed:
```

“‘z80 id:ch18_edge_detection_press_vs_hold input_flags: DB 0 ; current frame input_prev: DB 0 ; previous frame input_pressed: DB 0 ; newly pressed this frame (edges)

read_input_with_edges: ; Save previous state ld a, (input_flags) ld (input_prev), a

```
; Read current state
call read_input          ; updates input_flags

; Compute edges: pressed = current AND NOT previous
ld   a, (input_prev)
cpl                      ; 4T   invert previous
ld   b, a                ; 4T
ld   a, (input_flags)    ; 13T
and  b                   ; 4T   current AND NOT previous
ld   (input_pressed), a  ; 13T  = newly pressed this frame
ret


Now `input_pressed` has a 1 bit only for buttons that were *not* pressed last frame but *are*
shot actions (fire, jump, menu select).

### Agon Light 2: PS/2 Keyboard via MOS

The Agon reads its PS/2 keyboard through the MOS (Machine Operating System) API. The eZ80 does

The MOS system variable `sysvar_keyascii` (at address $0800 + offset) holds the ASCII code of

```z80 id:ch18_agon_light_2_ps_2_keyboard
; Agon: Read keyboard via MOS sysvar
; MOS sysvar_keyascii at IX+$05
read_input_agon:
    ld   a, (ix + $05)      ; read last key from MOS sysvars
    ; Map ASCII to input_flags
    cp   'o'
    jr   nz, .not_left
    set  INPUT_LEFT, d
.not_left:
    cp   'p'
    jr   nz, .not_right
    set  INPUT_RIGHT, d
.not_right:
    ; ... etc for Q, A, SPACE
    ld   a, d
    ld   (input_flags), a
    ret
```

The Agon also supports reading individual key states via VDU commands (VDU 23,0,$01,keycode), which return whether a specific key is currently held. This is closer to the Spectrum’s half-row approach and better suited for games that need simultaneous key detection. The MOS API handles the PS/2 protocol, scan code translation, and auto-repeat – none of which you need to worry about.
```

18.4 The Entity Structure

A game entity is anything that moves, animates, interacts, or needs per-frame updating: the player character, enemies, bullets, explosions, floating score numbers, power-ups. On the Z80, we represent each entity as a fixed-size block of bytes in memory.

Structure Layout

Here is the entity structure we will use throughout the game-dev chapters:

```
Offset  Size  Name        Description
------  ----  ----------  ----------------------------------------
 +0     2     x           X position, 8.8 fixed-point (high=pixel, low=subpixel)
 +2     1     y           Y position, pixel (0-191)
 +3     1     type        Entity type (0=inactive, 1=player, 2=enemy, 3=bullet,
↪  ...)
 +4     1     state       Entity state (0=idle, 1=active, 2=dying, 3=dead, ...)
 +5     1     anim_frame  Current animation frame index
 +6     1     dx          Horizontal velocity (signed, fixed-point fractional)
 +7     1     dy          Vertical velocity (signed, fixed-point fractional)
 +8     1     health      Hit points remaining
 +9     1     flags       Bit flags (see below)
------  ----
 10 bytes total per entity
```

Flag bits in the `flags` byte:

```
text id:ch18_structure_layout_2 Bit 0: ACTIVE      -- entity is alive and should
be updated/rendered Bit 1: VISIBLE     -- entity should be rendered (active but
invisible = logic only) Bit 2: COLLIDABLE  -- entity participates in collision
detection Bit 3: FACING_LEFT -- horizontal facing direction Bit 4: INVINCIBLE
-- temporary invulnerability (player after being hit) Bit 5: ON_GROUND   --
entity is standing on solid ground (set by physics) Bit 6-7: reserved
```

Why 10 Bytes?

Ten bytes is a deliberate choice. It is small enough that 16 entities occupy only 160 bytes – trivial in memory terms. More importantly, multiplying an entity index by 10 to find its offset is straightforward on the Z80:

```
z80 id:ch18_why_10_bytes ; Calculate entity address from index in A ; Input: A
= entity index (0-15) ; Output: HL = address of entity structure ; Destroys: DE
get_entity_addr:     ld   l, a               ; 4T     ld   h, 0              ;
7T     add  hl, hl             ; 11T  x2     ld   d, h              ; 4T     ld
e, l               ; 4T   DE = index x 2     add  hl, hl             ; 11T  x4
add  hl, hl             ; 11T  x8     add  hl, de             ; 11T  x8 + x2 =
x10     ld   de, entity_array   ; 10T     add  hl, de             ; 11T  HL =
entity_array + index * 10     ret                     ; 10T     ; Total: 94T
```

The multiplication by 10 uses the standard decomposition: 10 = 8 + 2. We compute index * 2, save it, compute index * 8, and add them together. No actual multiply instruction needed – just shifts (ADD HL,HL) and an addition.

If you chose a power-of-two size like 8 or 16 bytes per entity, the index calculation would be even simpler (three shifts for 8, four for 16). But 8 bytes is too cramped – you would lose either velocity or health, both of which matter. And 16 bytes wastes 6 bytes per entity on padding, which adds up: 16 entities x 6 wasted bytes = 96 bytes of dead space. On the Spectrum, every byte matters. Ten bytes is the right fit for the data we actually need.

Why 16-bit X but 8-bit Y?

The X position is 16-bit fixed-point (8.8 format): the high byte is the pixel column (0-255) and the low byte is a sub-pixel fraction for smooth movement. This is essential for horizontal scrolling games where the player moves at fractional-pixel speeds. A character moving at 1.5 pixels per frame with only integer coordinates would alternate between 1-pixel and 2-pixel steps, producing visible judder. With 8.8 fixed-point, the movement is smooth: add 0x0180 to X each frame and the pixel position advances 1, 2, 1, 2, 1, 2... in a pattern the eye perceives as a steady 1.5 pixels per frame.

The Y position is only 8 bits because the Spectrum's screen is 192 pixels tall – a single byte covers the full range. For a game with vertical scrolling, you would promote Y to 16-bit fixed-point as well, at the cost of one extra byte per entity.

The 8.8 Fixed-Point System

Fixed-point arithmetic was introduced in Chapter 4. Here is a quick recap of how it applies to entity movement:

"'z80 id:ch18_the_8_8_fixed_point_system ; Move entity right at velocity dx ; HL points to entity X (2 bytes: low=fractional, high=pixel) ; A = dx (signed velocity, treated as fractional byte) move_entity_x: ld c, (hl) ; 7T load X fractional part inc hl ; 6T ld b, (hl) ; 7T load X pixel part ; BC = 16-bit fixed-point X

```
ld   e, a               ; 4T   dx into E
; Sign-extend dx into DE
rla                      ; 4T   carry = sign bit
sbc  a, a               ; 4T   A = $FF if negative, $00 if positive
ld   d, a               ; 4T   DE = signed 16-bit dx

ex   de, hl             ; 4T
add  hl, de             ; 11T  new_X = old_X + dx (16-bit add)
; HL = new X position (fractional in L, pixel in H)

; Store back
ld   a, l               ; 4T
ld   (entity_x_lo), a   ; 13T  (self-modifying, or use IX)
ld   a, h               ; 4T
ld   (entity_x_hi), a   ; 13T
ret


The beauty of fixed-point: addition and subtraction are just regular 16-bit `ADD HL,DE` operat
pixel bits along.
```

```
---

## 18.5 The Entity Array

Entities live in a statically allocated array. No dynamic memory allocation, no linked lists,

```z80 id:ch18_the_entity_array
; Entity array: 16 entities, 10 bytes each = 160 bytes
MAX_ENTITIES    EQU  16
ENTITY_SIZE     EQU  10

entity_array:
    DS   MAX_ENTITIES * ENTITY_SIZE    ; 160 bytes, zeroed at init
```

Entity Slot Allocation

Slot 0 is always the player. Slots 1-8 are enemies. Slots 9-15 are projectiles and effects (bullets, explosions, score popups). This fixed partitioning simplifies the code: when you need to iterate over enemies for AI, you iterate slots 1-8. When a bullet needs spawning, you search slots 9-15. The player is always at a known address.

```
z80 id:ch18_entity_slot_allocation ; Fixed slot assignments SLOT_PLAYER      EQU
0 SLOT_ENEMY_FIRST EQU  1 SLOT_ENEMY_LAST  EQU  8 SLOT_PROJ_FIRST  EQU  9 SLOT_PROJ_LAST
EQU  15
```

Iterating Entities

The core update loop walks through every entity slot, checks the ACTIVE flag, and calls the appropriate update handler:

“‘z80 id:ch18_iterating_entities ; Update all active entities ; Total cost: ~2,500T for 16 entities (most inactive), up to ~8,000T (all active) update_entities: ld ix, entity_array ; 14T IX points to first entity ld b, MAX_ENTITIES ; 7T loop counter

.loop: ; Check if entity is active ld a, (ix + 9) ; 19T load flags byte (offset +9) bit 0, a ; 8T test ACTIVE flag jr z, .skip ; 12/7T skip if inactive

```
; Entity is active -- dispatch by type
ld   a, (ix + 3)        ; 19T  load type byte (offset +3)
; Jump table dispatch based on type
call update_by_type     ; ~200-500T depending on type
```

.skip: ; Advance IX to next entity ld de, ENTITY_SIZE ; 10T add ix, de ; 15T IX += 10 djnz .loop ; 13/8T ret

```
This uses IX as the entity pointer, which is convenient because IX-indexed addressing lets you
states versus 7 for `LD A,(HL)`. For the entity update loop, which runs 16 times per frame, th

### Update Dispatch by Type

Each entity type has its own update handler. We use the same jump-table technique as the game
```
```

```z80 id:ch18_update_dispatch_by_type
; Entity type constants
TYPE_INACTIVE  EQU  0
TYPE_PLAYER    EQU  1
TYPE_ENEMY     EQU  2
TYPE_BULLET    EQU  3
TYPE_EXPLOSION EQU  4

; Handler table (2 bytes per entry)
type_handlers:
    DW   update_inactive     ; type 0: no-op, should not be called
    DW   update_player       ; type 1
    DW   update_enemy        ; type 2
    DW   update_bullet       ; type 3
    DW   update_explosion    ; type 4

; Dispatch to type handler
; Input: A = entity type, IX = entity pointer
update_by_type:
    add  a, a               ; 4T   type * 2 (table entries are 2 bytes)
    ld   l, a               ; 4T
    ld   h, 0               ; 7T
    ld   de, type_handlers   ; 10T
    add  hl, de             ; 11T
    ld   e, (hl)            ; 7T
    inc  hl                 ; 6T
    ld   d, (hl)            ; 7T
    ex   de, hl             ; 4T
    jp   (hl)               ; 4T   jump to handler (RET will return to caller)
                            ; --- 64T dispatch overhead
```

Each handler receives IX pointing to the entity and can access all fields via indexed addressing. When the handler executes `RET`, it returns to the entity update loop, which advances to the next slot.

The Player Update Handler

Here is a typical player update – read input flags, apply movement, update animation:

“‘z80 id:ch18_the_player_update_handler ; Update player entity ; IX = entity pointer (slot 0) update_player: ; Read horizontal input ld a, (input_flags) ; 13T bit INPUT_RIGHT, a ; 8T jr z, .not_right ; 12/7T ; Move right: add dx to X ld a, 2 ; 7T dx = 2 subpixels per frame (~1 pixel/frame) add a, (ix + 0) ; 19T add to X fractional ld (ix + 0), a ; 19T jr nc, .no_carry_r ; 12/7T inc (ix + 1) ; 23T carry into X pixel .no_carry_r: res 3, (ix + 9) ; 23T clear FACING_LEFT flag jr .horiz_done ; 12T .not_right: bit INPUT_LEFT, a ; 8T jr z, .horiz_done ; 12/7T ; Move left: subtract dx from X ld a, (ix + 0) ; 19T load X fractional sub 2 ; 7T subtract dx ld (ix + 0), a ; 19T jr nc, .no_borrow_l ; 12/7T dec (ix + 1) ; 23T borrow from X pixel .no_borrow_l: set 3, (ix + 9) ; 23T set FACING_LEFT flag .horiz_done:

```
; Update animation frame (cycle every 8 frames)
ld   a, (ix + 5)        ; 19T  anim_frame
```

```
inc  a                  ; 4T
and  7                  ; 7T   wrap 0-7
ld   (ix + 5), a        ; 19T
ret
; Total: ~250-350T depending on input
```

This is deliberately simple. Chapter 19 will add gravity, jumping, and collision response. For

---

18.6 The Object Pool

Bullets, explosions, and particle effects are transient. A bullet exists for a fraction of a s
16 frames and vanishes. You could spawn these dynamically, but on the Z80, "dynamic" means sea

We already have the pool -- it is the entity array. Slots 9-15 are the projectile/effect pool.

Spawning a Bullet

```z80 id:ch18_spawning_a_bullet
; Spawn a bullet at position (B=x_pixel, C=y)
; moving in direction determined by player facing
; Returns: carry set if no free slot available
spawn_bullet:
    ld   ix, entity_array + (SLOT_PROJ_FIRST * ENTITY_SIZE)
    ld   d, SLOT_PROJ_LAST - SLOT_PROJ_FIRST + 1  ; 7 slots to check

.find_slot:
    ld   a, (ix + 9)        ; 19T  flags
    bit  0, a               ; 8T   ACTIVE?
    jr   z, .found          ; 12/7T found an inactive slot

    push de                 ; 11T  save loop counter (D)
    ld   de, ENTITY_SIZE    ; 10T  DE = 10 (D=0, E=10)
    add  ix, de             ; 15T  next slot
    pop  de                 ; 10T  restore loop counter
    dec  d                  ; 4T
    jr   nz, .find_slot     ; 12T

    ; No free slot -- set carry and return
    scf                      ; 4T
    ret

.found:
    ; Fill in the bullet entity
    ld   (ix + 0), 0        ; fractional X = 0
    ld   (ix + 1), b        ; pixel X = B
    ld   (ix + 2), c        ; Y = C
    ld   (ix + 3), TYPE_BULLET ; type
    ld   (ix + 4), 1        ; state = active
    ld   (ix + 5), 0        ; anim_frame = 0
```

```
    ld   (ix + 8), 1        ; health = 1 (dies on first collision)

    ; Set velocity based on player facing
    ld   a, (entity_array + 9)  ; player flags
    bit  3, a               ; FACING_LEFT?
    jr   z, .fire_right
    ld   (ix + 6), -4       ; dx = -4 (fast, leftward)
    jr   .set_flags
.fire_right:
    ld   (ix + 6), 4        ; dx = +4 (fast, rightward)
.set_flags:
    ld   (ix + 7), 0        ; dy = 0 (horizontal bullet)
    ld   (ix + 9), %00000111  ; flags: ACTIVE + VISIBLE + COLLIDABLE
    or   a                   ; clear carry (success)
    ret
```

Deactivating an Entity

When a bullet leaves the screen or an explosion finishes its animation, deactivation is a single instruction:

```
z80 id:ch18_deactivating_an_entity ; Deactivate entity at IX deactivate_entity:
ld   (ix + 9), 0        ; 19T  clear all flags (ACTIVE=0)     ret
```

That is it. Next frame, the update loop sees ACTIVE=0 and skips the slot. The slot is now available for the next `spawn_bullet` call to reuse.

Bullet Update Handler

“‘z80 id:ch18_bullet_update_handler ; Update a bullet entity ; IX = entity pointer update_bullet: ; Move horizontally ld a, (ix + 6) ; 19T dx ld e, a ; 4T ; Sign-extend rla ; 4T sbc a, a ; 4T ld d, a ; 4T DE = signed 16-bit dx

```
ld   l, (ix + 0)        ; 19T  X lo
ld   h, (ix + 1)        ; 19T  X hi
add  hl, de             ; 11T  new X
ld   (ix + 0), l        ; 19T
ld   (ix + 1), h        ; 19T

; Check screen bounds (0-255 pixel range)
ld   a, h               ; 4T   pixel X
or   a                   ; 4T
jr   z, .off_screen     ; boundary check: if X=0, leftward bullet exited
cp   248                ; 7T   near right edge?
jr   nc, .off_screen    ; past right boundary

; Still alive -- return
ret
```

.off_screen: ; Deactivate ld (ix + 9), 0 ; clear flags ret ; Total: ~170T active, ~190T when deactivating

Pool Sizing

```
Seven projectile slots (indices 9-15) might sound limited. In practice, it is more than enough

If you need more, expand the entity array. But be aware of the cost: each additional entity ad
states to the worst-case update loop (when active) and ~50 T-states even when inactive (the AC
two entities with all active would consume roughly 16,000 T-states in the update loop alone --

On the Agon, you can afford larger pools. With 360,000 T-states per frame and hardware sprite

---

## 18.7 Explosion and Effect Entities

Explosions, score popups, and particle effects use the same entity slots as bullets. The diffe
destruct.

```z80 id:ch18_explosion_and_effect_entities
; Update an explosion entity
; IX = entity pointer
update_explosion:
    ; Advance animation frame
    ld   a, (ix + 5)        ; 19T  anim_frame
    inc  a                   ; 4T
    cp   8                   ; 7T   8 frames of animation
    jr   nc, .done          ; 12/7T animation complete

    ld   (ix + 5), a        ; 19T  store new frame
    ret

.done:
    ; Animation complete -- deactivate
    ld   (ix + 9), 0        ; 19T  clear flags
    ret
```

To spawn an explosion when an enemy dies:

“‘z80 id:ch18_explosion_and_effect_entities_2 ; Spawn explosion at the enemy’s position ; IX currently points to the dying enemy spawn_explosion_at_entity: ld b, (ix + 1) ; enemy’s X pixel ld c, (ix + 2) ; enemy’s Y

```
; Find a free projectile/effect slot
push ix
ld   ix, entity_array + (SLOT_PROJ_FIRST * ENTITY_SIZE)
ld   d, SLOT_PROJ_LAST - SLOT_PROJ_FIRST + 1
```

.find: ld a, (ix + 9) bit 0, a jr z, .got_slot ld e, ENTITY_SIZE add ix, de dec d jr nz, .find pop ix ret ; no free slot – skip explosion

.got_slot: ld (ix + 0), 0 ; X fractional ld (ix + 1), b ; X pixel ld (ix + 2), c ; Y ld (ix + 3), TYPE_EXPLOSION ld (ix + 4), 1 ; state = active ld (ix + 5), 0 ; anim_frame = 0 ld (ix + 6), 0 ; dx = 0 (stationary) ld (ix + 7), 0 ; dy = 0 ld (ix + 8), 0 ; health = 0 (not collidable in a meaningful way) ld (ix + 9), %00000011 ; ACTIVE + VISIBLE, not COLLIDABLE pop ix ret
```

The pattern is always the same: find a free slot, fill in the structure, set the flags. The up
specific work. The deactivation clears the flags. The slot is reused next time something needs

---

18.8 Putting It All Together: The Game Skeleton

Here is the complete game skeleton that ties everything together. This is a compilable framewo

```z80 id:ch18_putting_it_all_together_the
    ORG  $8000

; ============================================================
; Constants
; ============================================================
MAX_ENTITIES    EQU  16
ENTITY_SIZE     EQU  10

STATE_TITLE     EQU  0
STATE_MENU      EQU  2
STATE_GAME      EQU  4
STATE_PAUSE     EQU  6
STATE_GAMEOVER  EQU  8

TYPE_INACTIVE   EQU  0
TYPE_PLAYER     EQU  1
TYPE_ENEMY      EQU  2
TYPE_BULLET     EQU  3
TYPE_EXPLOSION  EQU  4

INPUT_RIGHT     EQU  0
INPUT_LEFT      EQU  1
INPUT_DOWN      EQU  2
INPUT_UP        EQU  3
INPUT_FIRE      EQU  4

FLAG_ACTIVE     EQU  0
FLAG_VISIBLE    EQU  1
FLAG_COLLIDABLE EQU  2
FLAG_FACING_L   EQU  3

; ============================================================
; Entry point
; ============================================================
entry:
    di
    ld   sp, $C000          ; set stack (below banked memory on 128K)
                            ; NOTE: $FFFF is in banked page on 128K Spectrum,
                            ; which causes stack corruption during bank switches.
                            ; Use $C000 (or $BFFF) for 128K compatibility.
```

```
    im   1
    ei

    ; Clear entity array
    ld   hl, entity_array
    ld   de, entity_array + 1
    ld   bc, MAX_ENTITIES * ENTITY_SIZE - 1
    ld   (hl), 0
    ldir

    ; Start in title state
    ld   a, STATE_TITLE
    ld   (game_state), a

; ============================================================
; Main loop with state dispatch
; ============================================================
main_loop:
    halt                    ; sync to frame interrupt

    ; --- Border profiling: red = active processing ---
    ld   a, 2
    out  ($FE), a

    ; Dispatch to current state
    ld   a, (game_state)
    ld   l, a
    ld   h, 0
    ld   de, state_table
    add  hl, de
    ld   e, (hl)
    inc  hl
    ld   d, (hl)
    ex   de, hl
    jp   (hl)

; Called by each state handler when done
return_to_loop:
    ; --- Border black: idle ---
    xor  a
    out  ($FE), a
    jr   main_loop

; ============================================================
; State table
; ============================================================
state_table:
    DW   state_title
    DW   state_menu
    DW   state_game
    DW   state_pause
```

```
    DW   state_gameover

; ============================================================
; State: Title screen
; ============================================================
state_title:
    call read_input_with_edges
    ld   a, (input_pressed)
    bit  INPUT_FIRE, a
    jr   z, .wait
    ; Transition to game
    ld   a, STATE_GAME
    ld   (game_state), a
    call init_game
.wait:
    jp   return_to_loop

; ============================================================
; State: Game
; ============================================================
state_game:
    call read_input_with_edges

    ; Check pause
    ; (using 'P' key -- half-row $DF, bit 0 is P, but for simplicity
    ;  we check input_pressed bit 4 / FIRE as a toggle here)

    ; Update all entities
    call update_entities

    ; Render all visible entities
    call render_entities

    ; Update music
    ; call music_play         ; PT3 player -- see Chapter 11

    jp   return_to_loop

; ============================================================
; State: Pause (minimal)
; ============================================================
state_pause:
    call read_input_with_edges
    ld   a, (input_pressed)
    bit  INPUT_FIRE, a
    jr   z, .still_paused
    ld   a, STATE_GAME
    ld   (game_state), a
.still_paused:
    jp   return_to_loop
```

```
; ============================================================
; State: Game Over (minimal)
; ============================================================
state_gameover:
    call read_input_with_edges
    ld   a, (input_pressed)
    bit  INPUT_FIRE, a
    jr   z, .wait
    ld   a, STATE_TITLE
    ld   (game_state), a
.wait:
    jp   return_to_loop

; ============================================================
; State: Menu (minimal — expand for your game)
; ============================================================
state_menu:
    ; A full menu would display options and handle UP/DOWN/FIRE.
    ; For this skeleton, the menu simply transitions to the title.
    ; See Exercise 2 below for adding a real menu with item selection.
    jp   state_title

; ============================================================
; Init game: set up player and enemies
; ============================================================
init_game:
    ; Clear entity array
    ld   hl, entity_array
    ld   de, entity_array + 1
    ld   bc, MAX_ENTITIES * ENTITY_SIZE - 1
    ld   (hl), 0
    ldir

    ; Set up player (slot 0)
    ld   ix, entity_array
    ld   (ix + 0), 0        ; X fractional = 0
    ld   (ix + 1), 128      ; X pixel = 128 (centre)
    ld   (ix + 2), 160      ; Y = 160 (near bottom)
    ld   (ix + 3), TYPE_PLAYER
    ld   (ix + 4), 1        ; state = active
    ld   (ix + 5), 0        ; anim_frame
    ld   (ix + 6), 0        ; dx
    ld   (ix + 7), 0        ; dy
    ld   (ix + 8), 3        ; health = 3
    ld   (ix + 9), %00000111 ; ACTIVE + VISIBLE + COLLIDABLE

    ; Set up 8 enemies (slots 1-8) in a formation
    ld   ix, entity_array + ENTITY_SIZE   ; slot 1
    ld   b, 8               ; 8 enemies
    ld   c, 24              ; starting X pixel
```

```
.enemy_loop:
    ld   (ix + 0), 0        ; X fractional
    ld   (ix + 1), c        ; X pixel
    ld   (ix + 2), 32       ; Y = 32 (near top)
    ld   (ix + 3), TYPE_ENEMY
    ld   (ix + 4), 1        ; state = active
    ld   (ix + 5), 0        ; anim_frame
    ld   (ix + 6), 1        ; dx = 1 (moving right slowly)
    ld   (ix + 7), 0        ; dy = 0
    ld   (ix + 8), 1        ; health = 1
    ld   (ix + 9), %00000111 ; ACTIVE + VISIBLE + COLLIDABLE

    ; Advance to next slot and X position
    ld   de, ENTITY_SIZE
    add  ix, de
    ld   a, c
    add  a, 28              ; 28 pixels apart
    ld   c, a
    djnz .enemy_loop

    ret

; ============================================================
; Input system
; ============================================================
input_flags:      DB  0
input_prev:       DB  0
input_pressed:    DB  0

read_input_with_edges:
    ; Save previous
    ld   a, (input_flags)
    ld   (input_prev), a

    ; Read keyboard (QAOP + SPACE)
    ld   d, 0

    ; P key: half-row $DF, bit 0
    ld   bc, $DFFE
    in   a, (c)
    bit  0, a
    jr   nz, .no_right
    set  INPUT_RIGHT, d
.no_right:
    ; O key: half-row $DF, bit 1
    bit  1, a
    jr   nz, .no_left
    set  INPUT_LEFT, d
.no_left:

    ; Q key: half-row $FB, bit 0
```

```
    ld   b, $FB
    in   a, (c)
    bit  0, a
    jr   nz, .no_up
    set  INPUT_UP, d
.no_up:

    ; A key: half-row $FD, bit 0
    ld   b, $FD
    in   a, (c)
    bit  0, a
    jr   nz, .no_down
    set  INPUT_DOWN, d
.no_down:

    ; SPACE: half-row $7F, bit 0
    ld   b, $7F
    in   a, (c)
    bit  0, a
    jr   nz, .no_fire
    set  INPUT_FIRE, d
.no_fire:

    ld   a, d
    ld   (input_flags), a

    ; Compute edges
    ld   a, (input_prev)
    cpl
    ld   b, a
    ld   a, (input_flags)
    and  b
    ld   (input_pressed), a
    ret

; ============================================================
; Entity update loop
; ============================================================
update_entities:
    ld   ix, entity_array
    ld   b, MAX_ENTITIES

.loop:
    push bc
    ld   a, (ix + 9)        ; flags
    bit  FLAG_ACTIVE, a
    jr   z, .skip

    ld   a, (ix + 3)        ; type
    call update_by_type
```

```
.skip:
    ld   de, ENTITY_SIZE
    add  ix, de
    pop  bc
    djnz .loop
    ret

; ============================================================
; Type dispatch
; ============================================================
type_handlers:
    DW   .nop_handler       ; TYPE_INACTIVE
    DW   update_player
    DW   update_enemy
    DW   update_bullet
    DW   update_explosion

update_by_type:
    add  a, a
    ld   l, a
    ld   h, 0
    ld   de, type_handlers
    add  hl, de
    ld   e, (hl)
    inc  hl
    ld   d, (hl)
    ex   de, hl
    jp   (hl)

.nop_handler:
    ret

; ============================================================
; Player update
; ============================================================
update_player:
    ld   a, (input_flags)
    bit  INPUT_RIGHT, a
    jr   z, .not_right
    ld   a, (ix + 0)
    add  a, 2               ; move right (subpixel)
    ld   (ix + 0), a
    jr   nc, .x_done_r
    inc  (ix + 1)
.x_done_r:
    res  FLAG_FACING_L, (ix + 9)
    jr   .horiz_done
.not_right:
    bit  INPUT_LEFT, a
    jr   z, .horiz_done
    ld   a, (ix + 0)
```

```
    sub  2                   ; move left (subpixel)
    ld   (ix + 0), a
    jr   nc, .x_done_l
    dec  (ix + 1)
.x_done_l:
    set  FLAG_FACING_L, (ix + 9)
.horiz_done:

    ; Fire bullet on press (edge-detected)
    ld   a, (input_pressed)
    bit  INPUT_FIRE, a
    jr   z, .no_fire
    ld   b, (ix + 1)        ; player X pixel
    ld   c, (ix + 2)        ; player Y
    call spawn_bullet
.no_fire:

    ; Animate
    ld   a, (ix + 5)
    inc  a
    and  7
    ld   (ix + 5), a
    ret

; ============================================================
; Enemy update (simple patrol)
; ============================================================
update_enemy:
    ; Move by dx
    ld   a, (ix + 6)        ; dx
    add  a, (ix + 1)        ; add to X pixel
    ld   (ix + 1), a

    ; Bounce at screen edges
    cp   240
    jr   c, .no_bounce_r
    ld   (ix + 6), -1       ; reverse direction
    jr   .bounce_done
.no_bounce_r:
    cp   8
    jr   nc, .bounce_done
    ld   (ix + 6), 1        ; reverse direction
.bounce_done:

    ; Animate
    ld   a, (ix + 5)
    inc  a
    and  3                   ; 4-frame animation cycle
    ld   (ix + 5), a
    ret
```

```
; ============================================================
; Bullet update
; ============================================================
update_bullet:
    ld   a, (ix + 6)        ; dx
    add  a, (ix + 1)        ; add to X pixel (simplified: integer movement)
    ld   (ix + 1), a

    ; Off screen?
    cp   248
    jr   nc, .deactivate
    or   a
    jr   z, .deactivate
    ret

.deactivate:
    ld   (ix + 9), 0        ; clear all flags
    ret

; ============================================================
; Explosion update
; ============================================================
update_explosion:
    ld   a, (ix + 5)        ; anim_frame
    inc  a
    cp   8                   ; 8 frames
    jr   nc, .done
    ld   (ix + 5), a
    ret
.done:
    ld   (ix + 9), 0
    ret

; ============================================================
; Spawn bullet
; ============================================================
spawn_bullet:
    ; B = x pixel, C = y
    push ix
    ld   ix, entity_array + (9 * ENTITY_SIZE)   ; first projectile slot
    ld   d, 7               ; 7 slots to search

.find:
    ld   a, (ix + 9)
    bit  FLAG_ACTIVE, a
    jr   z, .found
    push de                  ; save loop counter in D
    ld   de, ENTITY_SIZE     ; DE = 10 (D=0, E=10)
    add  ix, de
    pop  de                  ; restore loop counter
    dec  d
```

```
    jr   nz, .find
    ; No free slot
    pop  ix
    scf
    ret

.found:
    ld   (ix + 0), 0
    ld   (ix + 1), b
    ld   (ix + 2), c
    ld   (ix + 3), TYPE_BULLET
    ld   (ix + 4), 1
    ld   (ix + 5), 0
    ld   (ix + 7), 0        ; dy = 0

    ; Direction from player facing
    ld   a, (entity_array + 9)   ; player flags
    bit  FLAG_FACING_L, a
    jr   z, .right
    ld   (ix + 6), -4       ; dx = -4
    jr   .dir_done
.right:
    ld   (ix + 6), 4        ; dx = +4
.dir_done:
    ld   (ix + 8), 1        ; health = 1
    ld   (ix + 9), %00000111 ; ACTIVE + VISIBLE + COLLIDABLE

    pop  ix
    or   a                   ; clear carry
    ret

; ============================================================
; Render entities (stub -- see Chapter 16 for sprite rendering)
; ============================================================
render_entities:
    ; In a real game, this iterates visible entities and draws sprites.
    ; See Chapter 16 for OR+AND masked sprites, pre-shifted sprites,
    ; and the dirty-rectangle system.
    ; For this skeleton, we use a minimal attribute-block renderer.
    ld   ix, entity_array
    ld   b, MAX_ENTITIES

.loop:
    push bc
    ld   a, (ix + 9)
    bit  FLAG_VISIBLE, a
    jr   z, .skip

    ; Draw a 1-character coloured block at entity position.
    ; For real sprite rendering, see Chapter 16 (OR+AND masks,
    ; pre-shifted sprites, compiled sprites, dirty rectangles).
```

```
    ld   a, (ix + 1)        ; X pixel
    rrca                     ; /2
    rrca                     ; /4
    rrca                     ; /8 = character column
    and  $1F                 ; mask to 0-31
    ld   e, a
    ld   a, (ix + 2)        ; Y pixel
    rrca
    rrca
    rrca
    and  $1F                 ; character row (0-23)
    ; Compute attribute address: $5800 + row*32 + col
    ld   l, a
    ld   h, 0
    add  hl, hl             ; row * 2
    add  hl, hl             ; row * 4
    add  hl, hl             ; row * 8
    add  hl, hl             ; row * 16
    add  hl, hl             ; row * 32
    ld   d, 0
    add  hl, de             ; + column
    ld   de, $5800
    add  hl, de             ; HL = attribute address

    ; Colour by type
    ld   a, (ix + 3)        ; type
    add  a, a               ; type * 2 (crude colour mapping)
    or   %01000000           ; BRIGHT bit
    ld   (hl), a            ; write attribute

.skip:
    ld   de, ENTITY_SIZE
    add  ix, de
    pop  bc
    djnz .loop
    ret

; ============================================================
; Data
; ============================================================
game_state:     DB   STATE_TITLE

entity_array:
    DS   MAX_ENTITIES * ENTITY_SIZE, 0
```

This skeleton compiles, runs, and does something visible: coloured blocks move across the attribute grid. The player block responds to QAOP controls. Pressing SPACE spawns bullets that fly across the screen. Enemies bounce between the screen edges. When a bullet exits the screen, its slot frees up for the next shot.

It is ugly – attribute blocks instead of sprites, no scrolling, no sound. But the architecture is complete. Every piece from this chapter is present and wired

together: the HALT-driven main loop, the state machine dispatch, the input reader with edge detection, the entity array with per-type update handlers, and the object pool for projectiles. Chapters 16 and 17 provide the rendering. Chapter 19 provides the physics and collisions. Chapter 11 provides the music. This skeleton is where they all plug in.

18.9 Agon Light 2: The Same Architecture, More Room

The Agon Light 2 uses the same fundamental game loop structure. The eZ80 runs Z80 code natively (in Z80-compatible mode or ADL mode), so the HALT-based main loop, the state machine, the entity system, and the input logic all translate directly.

The key differences:

Frame sync. The Agon uses MOS's `waitvblank` call (RST $08, function $1E) instead of `HALT` for frame synchronisation. The VDP generates the vertical blank signal and the MOS API exposes it.

Input. Keyboard reading goes through MOS system variables rather than direct port I/O. The half-row matrix does not exist – the PS/2 keyboard is handled by the ESP32 VDP. The input abstraction layer we built (everything funnels into `input_flags`) means the rest of the game code does not care about the difference.

Entity budget. With ~360,000 T-states per frame and hardware sprite rendering, the entity update loop is no longer a bottleneck. You could update 64 entities with complex AI and still use under 10% of the frame budget. The limiting factor on the Agon is VDP sprite count per scanline (typically 16-32 hardware sprites visible on the same line) rather than CPU time.

Rendering. The Agon's VDP handles sprite rendering. Instead of manually blitting pixels into screen memory (Chapter 16's six methods), you issue VDU commands to position hardware sprites. The CPU cost per sprite drops from ~1,200 T-states (OR+AND blit on Spectrum) to ~50-100 T-states (sending a VDU position command). This frees enormous CPU time for game logic.

Memory. The Agon has 512KB of flat memory – no banking, no contended regions. Your entity array, lookup tables, sprite data, level maps, and music can all coexist without the bank-switching gymnastics that Chapter 15 describes for the Spectrum 128K.

The practical takeaway: on the Agon, this chapter's architecture scales effortlessly. More entities, more complex state machines, more AI logic – none of it threatens the frame budget. The discipline of counting every T-state still matters (it is good engineering), but the constraints that force agonising trade-offs on the Spectrum simply do not apply.

18.10 Design Decisions and Trade-Offs

Fixed vs Variable Frame Rate

This chapter's main loop assumes a fixed 50 fps frame rate: do everything in one frame, or drop. The alternative is a variable time step: measure how long the frame took and scale all movement by delta-time. Variable time steps are standard in modern game engines but add complexity on the Z80 – you need a frame timer, multiplication by delta in every movement calculation, and careful handling of physics stability at variable rates.

For Spectrum games, fixed 50 fps is almost universally the right choice. The hardware is deterministic, the frame budget is predictable, and the simplicity of fixed-step physics (everything moves by constant amounts each frame) eliminates an entire category of bugs. If your game drops below 50 fps, the answer is to optimise until it does not – not to add a variable time step.

On the Agon, with its larger budget, you are even less likely to need variable timing. Fix the frame rate at 50 or 60 fps and keep life simple.

Entity Size: Lean vs Generous

Our 10-byte entity structure is lean. Some commercial Spectrum games used 16 or even 32 bytes per entity, storing additional fields like previous position (for dirty-rectangle erasure), sprite address, collision box dimensions, AI timer, and more.

The trade-off is iteration speed versus field access. Our 16-entity array takes 160 bytes and the full update loop runs in ~8,000 T-states. A 32-byte structure with 16 entities takes 512 bytes (still small) but the iteration overhead grows because IX advances by 32 each step, and indexed accesses to fields at high offsets like `(IX+28)` take the same 19 T-states but are harder to keep track of.

If you need more per-entity data, consider splitting the structure: a compact "hot" array (position, type, flags – the fields touched every frame) and a parallel "cold" array (sprite address, AI state, score value – fields accessed only when needed). This is the same structure-of-arrays versus array-of-structures trade-off that modern game engines face, applied at the Z80 scale.

When to Use HL Instead of IX

IX-indexed addressing is convenient but expensive: 19 T-states per access versus 7 for `(HL)`. In the update loop (called 16 times per frame), the IX overhead is acceptable – 16 x 12 extra T-states per access = 192 T-states, negligible.

But in the rendering loop, where you might touch 4-6 entity fields for each of 8 visible sprites, the cost adds up. The technique: at the start of the render pass for each entity, copy the fields you need into registers:

```
z80 id:ch18_when_to_use_hl_instead_of_ix     ; Copy entity fields to registers
for fast rendering     ld   l, (ix + 0)        ; 19T  X lo     ld   h, (ix + 1)
; 19T  X hi     ld   c, (ix + 2)        ; 19T  Y     ld   a, (ix + 5)        ;
19T  anim_frame     ; Now render using H (X pixel), C (Y), A (frame)     ; All
subsequent accesses are register-to-register: 4T each
```

Four IX accesses at 19T = 76T up front, then the entire render routine uses 4T register accesses instead of 19T IX accesses. If the render routine touches those fields 10 times, you save (19-4) x 10 - 76 = 74 T-states per entity. Small, but over 8 entities per frame, that is 592 T-states – enough to draw another half-sprite.

Summary

- The **game loop** is `HALT -> Input -> Update -> Render -> repeat`. The `HALT` instruction synchronises to the frame interrupt, giving you exactly one frame's worth of T-states (roughly 64,000 on a Pentagon, 360,000 on the Agon).
- A **state machine** with a jump table of handler pointers (`DW state_title`, `DW state_game`, etc.) organises the flow from title screen through gameplay to game over. Dispatch costs 73 T-states regardless of state count – constant time, clean scaling.
- **Input reading** on the Spectrum uses `IN A,(C)` to poll keyboard half-rows through port `$FE`. Five keys (QAOP + SPACE) cost roughly 220 T-states to read. The Kempston joystick is a single 11T port read. Edge detection (press vs hold) uses XOR between current and previous frames.
- The **entity structure** is 10 bytes: X (16-bit fixed-point), Y, type, state, anim_frame, dx, dy, health, flags. Sixteen entities occupy 160 bytes. Multiplication by 10 for index-to-address conversion uses the decomposition 10 = 8 + 2.
- The **entity array** is statically allocated with fixed slot assignments: slot 0 for the player, slots 1-8 for enemies, slots 9-15 for projectiles and effects. Iteration checks the ACTIVE flag and dispatches to per-type handlers via a second jump table.
- The **object pool** is the entity array itself. Spawning sets fields and the ACTIVE flag. Deactivation clears the flags. Free-slot search is a linear scan of the relevant slot range. Seven projectile slots handle typical fire rates without the player noticing missed spawns.
- **IX-indexed addressing** is convenient for entity field access (19T per access) but expensive in inner loops. Copy fields to registers at the start of rendering for 4T access throughout.
- The Agon Light 2 uses the same architecture with more headroom. MOS `waitvblank` replaces `HALT`, PS/2 keyboard replaces half-row scanning, hardware sprites replace CPU blitting. The entity update loop is no longer the bottleneck.
- The practical skeleton in this chapter runs a state machine, 16 entities (1 player + 8 enemies + 7 bullet/effect slots), input with edge detection, per-type update handlers, and a minimal attribute-block renderer. It is the chassis that Chapters 16 (sprites), 17 (scrolling), 19 (collisions), and 11 (sound) plug into.

Try It Yourself

1. **Build the skeleton.** Compile the game skeleton from section 18.8 and run it in an emulator. Use QAOP to move the player block and SPACE to fire. Watch the coloured blocks move. Add border-colour profiling (Chapter 1) to see how much frame budget is used.
2. **Add a sixth state.** Implement STATE_SHOP between Menu and Game. The shop screen should display three items and let the player choose one with UP/DOWN and FIRE. This exercises the state machine – add the constant, the table entry, the handler, and the transition logic.
3. **Expand the entity count.** Increase MAX_ENTITIES to 32, add 16 more enemies, and measure the frame budget impact with border profiling. At what entity count does the update loop start threatening 50 fps?
4. **Implement Kempston support.** Add the Kempston joystick reader and combine it with keyboard input using OR. Test in an emulator that supports Kempston emulation (Fuse: Options -> Joysticks -> Kempston).
5. **Split hot and cold data.** Create a second "cold" array with 4 bytes per entity (sprite address, AI timer, AI state, score value). Modify the update loop to access cold data only when the entity type requires it (enemies for AI, not bullets). Measure the T-state savings.

Next: Chapter 19 – Collisions, Physics, and Enemy AI. We will add AABB collision detection, gravity, jumping, and four enemy behaviour patterns to the skeleton from this chapter.

Sources: ZX Spectrum keyboard matrix and port layout from Sinclair Research technical documentation; Kempston joystick interface specification; fixed-point arithmetic techniques from Chapter 4 (Dark/X-Trade, Spectrum Expert #01, 1997); border-colour profiling from Chapter 1; sprite rendering methods referenced from Chapter 16; AY music integration referenced from Chapter 11; Agon Light 2 MOS API documentation (Bernardo Kastrup, agon-light.com)

Chapter 19: Collisions, Physics, and Enemy AI

> "Every game is a lie. Physics is faked. Intelligence is a table lookup. The player never notices because the lies are told at 50 frames per second."

In Chapter 18, we built a game loop, an entity system that tracks sixteen objects, and an input handler. But right now our player walks through walls, floats above the ground, and the enemies stand still. A game with no collisions is a screensaver. A game with no physics is a slide puzzle. A game with no AI is a sandbox with nothing to push back.

This chapter adds the three systems that turn a tech demo into a game: collision detection, physics, and enemy AI. All three share a design philosophy: fake it well enough, fast enough, and nobody will know the difference. We build on the entity structure from Chapter 18 – the 16-byte record with X/Y positions in 8.8 fixed-point, velocity in dx/dy, type, state, and flags.

Part 1: Collision Detection

AABB: The Only Shape You Need

Axis-Aligned Bounding Boxes. Every entity gets a rectangle defined by its position and dimensions: left edge, right edge, top edge, bottom edge. Two rectangles overlap if and only if all four of these conditions are true:

1. A's left edge is less than B's right edge
2. A's right edge is greater than B's left edge
3. A's top edge is less than B's bottom edge
4. A's bottom edge is greater than B's top edge

If any one of these conditions fails, the boxes do not overlap. This is the **early exit** that makes AABB fast: on average, most entity pairs are *not* colliding, so most checks bail out after one or two comparisons rather than doing all four.

"'mermaid id:ch19_aabb_the_only_shape_you_need graph TD START["Check collisionbetween Entity A and Entity B"] –> X1{"A.left < B.right?(A.x < B.x + B.width)"} X1 – No –> MISS["No collision(clear carry, return)"] X1 – Yes –> X2{"A.right > B.left?(A.x + A.width > B.x)"} X2 – No –> MISS X2 – Yes –> Y1{"A.top < B.bottom?(A.y < B.y + B.height)"} Y1 – No –> MISS Y1 – Yes –> Y2{"A.bottom > B.top?(A.y + A.height > B.y)"} Y2 – No –> MISS Y2 – Yes –> HIT["COLLISION!(set carry, return)"]

```
style MISS fill:#dfd,stroke:#393
style HIT fill:#fdd,stroke:#933
style X1 fill:#eef,stroke:#339
style X2 fill:#eef,stroke:#339
style Y1 fill:#fee,stroke:#933
style Y2 fill:#fee,stroke:#933
```

> **Early exit saves cycles:** Most entity pairs are far apart. The first X-overlap test rejects them in ~91 T-states. Only pairs that pass all four tests (worst case: ~2 states) are actual collisions. Test horizontal overlap first in side-scrollers -- entities are

On the Z80, we store entity positions as 8.8 fixed-point values, but for collision detection w level precision is more than enough. Here is a complete AABB collision routine:

```z80 id:ch19_aabb_the_only_shape_you_need_2
; check_aabb -- Test whether two entities overlap
;
; Input:  IX = pointer to entity A
;         IY = pointer to entity B
; Output: Carry set if collision, clear if no collision
;
; Entity structure offsets (from Chapter 18):
;   +0  x_frac    (low byte of 8.8 X position)
;   +1  x_int     (high byte -- the pixel X coordinate)
;   +2  y_frac
;   +3  y_int
;   +4  type
;   +5  state
;   +6  anim_frame
;   +7  dx_frac
;   +8  dx_int
;   +9  dy_frac
;   +10 dy_int
;   +11 health
;   +12 flags
;   +13 width     (bounding box width in pixels)
;   +14 height    (bounding box height in pixels)
;   +15 (reserved)
;
; Cost: 91-270 T-states (Pentagon), depending on early exit
; Average case (no collision): ~120 T-states

check_aabb:
    ; --- Test 1: A.left < B.right ---
    ; A.left  = A.x_int
    ; B.right = B.x_int + B.width
    ld   a, (iy+1)        ; 19T  B.x_int
    add  a, (iy+13)       ; 19T  + B.width = B.right
    ld   b, a             ; 4T   B = B.right (save for test 2)
    ld   a, (ix+1)        ; 19T  A.x_int = A.left
    cp   b                ; 4T   A.left - B.right
```

```
    jr   nc, .no_collision ; 12/7T  if A.left >= B.right, no collision
                           ; --- early exit: 91T (taken, incl. .no_collision) ---

    ; --- Test 2: A.right > B.left ---
    ; A.right = A.x_int + A.width
    ; B.left  = B.x_int
    add  a, (ix+13)        ; 19T  A.x_int + A.width = A.right
    ld   b, (iy+1)         ; 19T  B.x_int = B.left
    cp   b                 ; 4T   A.right - B.left (we need A.right > B.left)
    jr   c, .no_collision  ; 12/7T  if A.right < B.left, no collision
    jr   z, .no_collision  ; 12/7T  if A.right = B.left, touching but not overlapping

    ; --- Test 3: A.top < B.bottom ---
    ld   a, (iy+3)         ; 19T  B.y_int
    add  a, (iy+14)        ; 19T  + B.height = B.bottom
    ld   b, a              ; 4T
    ld   a, (ix+3)         ; 19T  A.y_int = A.top
    cp   b                 ; 4T   A.top - B.bottom
    jr   nc, .no_collision ; 12/7T  if A.top >= B.bottom, no collision

    ; --- Test 4: A.bottom > B.top ---
    add  a, (ix+14)        ; 19T  A.y_int + A.height = A.bottom
    ld   b, (iy+3)         ; 19T  B.y_int = B.top
    cp   b                 ; 4T   A.bottom - B.top
    jr   c, .no_collision  ; 12/7T
    jr   z, .no_collision  ; 12/7T

    ; All four tests passed -- collision detected
    scf                    ; 4T   set carry flag
    ret                    ; 10T

.no_collision:
    or   a                 ; 4T   clear carry flag
    ret                    ; 10T
```

The IX/IY indexed addressing is convenient but expensive – 19 T-states per access versus 7 for `ld a, (hl)`. For a game with 8 enemies and 7 bullets, it is acceptable. Worst case (all four tests pass, collision detected): approximately 270 T-states. Best case (first test fails): approximately 91 T-states. For 8 enemies checked against the player, the average case is about 8 x 120 = 960 T-states – 1.3% of the Pentagon frame budget. Collisions are cheap.

Overflow warning: The `ADD A, (ix+13)` instructions compute `x + width` in an 8-bit register. If an entity is positioned at X=240 with width=24, the result wraps around to 8, producing incorrect comparisons. Ensure that entity positions are clamped so that `x + width` and `y + height` never exceed 255 – typically by limiting the play area to leave a margin at the right and bottom edges. Alternatively, promote the comparison to 16-bit arithmetic at the cost of additional instructions.

Ordering the Tests for Fastest Rejection

The order matters. In a side-scroller, entities far apart horizontally are the common case. Testing horizontal overlap first rejects these after two comparisons. You can go further with a quick pre-rejection:

“‘z80 id:ch19_ordering_the_tests_for ; Quick X-distance rejection before calling check_aabb ; If the horizontal distance between entities exceeds ; MAX_WIDTH (the widest entity), they cannot collide.

```
ld   a, (ix+1)        ; 19T  A.x_int
sub  (iy+1)           ; 19T  - B.x_int
jr   nc, .pos_dx      ; 12/7T
neg                   ; 8T   absolute value
```

.pos_dx: cp MAX_WIDTH ; 7T widest possible entity jr nc, .skip ; 12/7T too far apart, skip AABB check call check_aabb ; only test close pairs .skip:

```
This pre-rejection costs about 60 T-states, saving the 82+ T-states of the full AABB check. On
3 enemies are close enough to need the full test.

### Tile Collisions: The Tilemap as a Collision Surface

In a platformer, the player collides with the world -- floors, walls, ceilings, spikes. We use

Assume a 32x24 tilemap with 8x8 pixel tiles (the natural Spectrum character grid):

```z80 id:ch19_tile_collisions_the_tilemap
; tile_at -- Look up the tile type at a pixel position
;
; Input:  B = pixel X, C = pixel Y
; Output: A = tile type (0=empty, 1=solid, 2=hazard, 3=ladder, etc.)
;
; Map is 32 columns wide, stored row-major at 'tilemap'
; Cost: ~182 T-states (Pentagon)

tile_at:
    ld   a, c             ; 4T   pixel Y
    srl  a                ; 8T   /2
    srl  a                ; 8T   /4
    srl  a                ; 8T   /8 = tile row
    ld   l, a             ; 4T

    ; Multiply row by 32 (shift left 5)
    ld   h, 0             ; 7T
    add  hl, hl           ; 11T  *2
    add  hl, hl           ; 11T  *4
    add  hl, hl           ; 11T  *8
    add  hl, hl           ; 11T  *16
    add  hl, hl           ; 11T  *32

    ld   a, b             ; 4T   pixel X
    srl  a                ; 8T   /2
```
```

```
    srl  a               ; 8T   /4
    srl  a               ; 8T   /8 = tile column
    ld   e, a            ; 4T
    ld   d, 0            ; 7T
    add  hl, de          ; 11T  row*32 + column = tile index

    ld   de, tilemap     ; 10T
    add  hl, de          ; 11T  absolute address

    ld   a, (hl)         ; 7T   tile type
    ret                   ; 10T
```

Now check the corners and edges of the entity against the tilemap:

“‘z80 id:ch19_tile_collisions_the_tilemap_2 ; check_player_tiles – Check player against tilemap ; ; Input: IX = player entity ; Output: Updates player position/velocity based on tile collisions ; ; We check up to 6 points around the player’s bounding box, ; but bail out as soon as we find a solid tile.

check_player_tiles: ; — Check below (feet) — ; Bottom-left corner of player ld b, (ix+1) ; 19T x_int ld a, (ix+3) ; 19T y_int add a, (ix+14) ; 19T + height = bottom edge ld c, a ; 4T call tile_at ; 17T+body cp TILE_SOLID ; 7T jr z, .on_ground ; 12/7T

```
; Bottom-right corner
ld   a, (ix+1)        ; 19T  x_int
add  a, (ix+13)       ; 19T  + width
dec  a                ; 4T   -1 (rightmost pixel of entity)
ld   b, a
ld   a, (ix+3)
add  a, (ix+14)
ld   c, a
call tile_at
cp   TILE_SOLID
jr   z, .on_ground

; Not standing on solid ground -- apply gravity
jr   .in_air
```

.on_ground: ; Snap Y to top of tile, clear vertical velocity ld a, c ; bottom edge Y and %11111000 ; align to tile boundary (clear low 3 bits) sub (ix+14) ; subtract height to get top-left Y ld (ix+3), a ; snap y_int xor a ld (ix+9), a ; dy_frac = 0 ld (ix+10), a ; dy_int = 0 set 0, (ix+12) ; set “on_ground” flag in flags byte jr .check_walls

.in_air: res 0, (ix+12) ; clear “on_ground” flag

.check_walls: ; — Check right (wall) — ld a, (ix+1) add a, (ix+13) ; right edge ld b, a ld a, (ix+3) add a, 4 ; check midpoint vertically ld c, a call tile_at cp TILE_SOLID jr nz, .check_left

```
; Push out left: snap X to left edge of tile
ld   a, b
and  %11111000
dec  a
sub  (ix+13)
inc  a
```

```
ld   (ix+1), a
xor  a
ld   (ix+7), a         ; dx_frac = 0
ld   (ix+8), a         ; dx_int = 0
```

.check_left: ; — Check left (wall) — ld b, (ix+1) ; left edge ld a, (ix+3) add a, 4 ld c, a call tile_at cp TILE_SOLID jr nz, .check_ceiling

```
; Push out right: snap X to right edge of tile + 1
ld   a, b
and  %11111000
add  a, 8
ld   (ix+1), a
xor  a
ld   (ix+7), a
ld   (ix+8), a
```

.check_ceiling: ; — Check above (head) — ld b, (ix+1) ld c, (ix+3) ; top edge call tile_at cp TILE_SOLID ret nz

```
; Hit ceiling: push down, zero vertical velocity
ld   a, c
and  %11111000
add  a, 8              ; bottom of ceiling tile
ld   (ix+3), a
xor  a
ld   (ix+9), a
ld   (ix+10), a
ret
```

```
The critical insight: point-in-tile lookups are O(1) array accesses. Each `tile_at` call costs
states. The entire tile collision system (checking feet, head, left, and right) costs roughly
1,200 T-states per entity, regardless of map size.

### Sliding Collision Response

When the player hits a wall while moving diagonally, they should *slide*, not stop dead. Resol

1. Apply horizontal velocity. Check horizontal tile collisions. If blocked, push out and zero
2. Apply vertical velocity. Check vertical tile collisions. If blocked, push out and zero vert

This is exactly what `check_player_tiles` does -- each axis is handled separately. Diagonal mo
controlled), then Y (gravity). Experiment with both orders and feel the difference.

---

## Part 2: Physics

What we are building is not a rigid-body simulation -- it is a small set of rules that produce

### Gravity: Falling Convincingly

Every frame, add a constant to the entity's vertical velocity:
```

```z80 id:ch19_gravity_falling_convincingly
; apply_gravity -- Add gravity to an entity's vertical velocity
;
; Input:  IX = entity pointer
; Output: dy updated (8.8 fixed-point, positive = downward)
;
; GRAVITY_FRAC and GRAVITY_INT define the gravity constant
; in 8.8 fixed-point. A good starting value: 0.25 per frame
; = $0040 (INT=0, FRAC=64, i.e. 64/256 = 0.25 pixels/frame^2)
;
; Cost: ~50 T-states (Pentagon)

GRAVITY_FRAC equ 40h     ; 0.25 pixels/frame^2 (fractional part)
GRAVITY_INT  equ 00h     ; (integer part)
MAX_FALL_INT equ 04h     ; terminal velocity: 4 pixels/frame

apply_gravity:
    ; Skip if entity is on the ground
    bit  0, (ix+12)       ; 20T  check on_ground flag
    ret  nz               ; 11/5T  on ground -- no gravity

    ; dy += gravity (16-bit fixed-point add)
    ld   a, (ix+9)        ; 19T  dy_frac
    add  a, GRAVITY_FRAC  ; 7T
    ld   (ix+9), a        ; 19T

    ld   a, (ix+10)       ; 19T  dy_int
    adc  a, GRAVITY_INT   ; 7T   add with carry from frac
    ld   (ix+10), a       ; 19T

    ; Clamp to terminal velocity
    cp   MAX_FALL_INT     ; 7T
    ret  c                ; 11/5T  below terminal velocity, done
    ld   (ix+10), MAX_FALL_INT ; 19T  clamp integer part
    xor  a                ; 4T
    ld   (ix+9), a        ; 19T  zero fractional part (exact clamp)
    ret                    ; 10T
```

The fixed-point representation from Chapter 4 is doing the heavy lifting here. Gravity is 0.25 pixels per frame squared – a value that would be impossible to represent with integer arithmetic. In 8.8 fixed-point, it is simply `$0040`. Each frame, `dy` grows by 0.25. After 4 frames, the entity is falling at 1 pixel per frame. After 16 frames, it is falling at 4 pixels per frame (terminal velocity). The acceleration curve feels natural because it *is* natural – constant acceleration is just velocity incrementing linearly.

The terminal velocity clamp prevents entities from falling so fast they skip through floors (the "tunnelling" problem). A maximum fall speed of 4 pixels per frame means the entity can never move more than half the tile height in one frame, so tile collision checks will always catch it.

Why Fixed-Point Matters Here

Without fixed-point, gravity is either 0 or 1 pixel per frame – float or stone, nothing in between. 8.8 fixed-point gives you 256 values between each integer. $0040 (0.25) produces a gentle arc. $0080 (0.5) feels heavy. $0020 (0.125) feels like a moon jump. Tuning these constants is where your game finds its character. If the fixed-point fundamentals are hazy, revisit Chapter 4.

Jump: The Anti-Gravity Impulse

Jumping is the simplest physics operation in the game: set the vertical velocity to a large negative value (upward). Gravity will decelerate it, bring it to zero at the apex, and pull it back down. The jump arc is a natural parabola – no explicit arc calculation needed.

“‘z80 id:ch19_jump_the_anti_gravity_impulse ; try_jump – Initiate a jump if the player is on the ground ; ; Input: IX = player entity ; Output: dy set to -jump_force if on ground ; ; JUMP_FORCE defines the initial upward velocity in 8.8 fixed-point. ; A good starting value: -3.5 pixels/frame = $FC80 ; (INT = $FC = -4 signed, FRAC = $80 = +0.5, so -4 + 0.5 = -3.5) ; ; Cost: ~50 T-states (Pentagon)

JUMP_FRAC equ 80h ; fractional part of jump force JUMP_INT equ 0FCh ; integer part (-4 signed + 0.5 frac = -3.5)

try_jump: ; Must be on ground to jump bit 0, (ix+12) ; 20T on_ground flag ret z ; 11/5T in air – cannot jump

```
; Set upward velocity
ld   (ix+9), JUMP_FRAC  ; 19T  dy_frac
ld   (ix+10), JUMP_INT  ; 19T  dy_int = -3.5 (upward)

; Clear on_ground flag
res  0, (ix+12)       ; 23T

; (Optional: play jump sound effect here)
ret                    ; 10T


With gravity at 0.25/frame^2 and jump force at -3.5/frame, the player rises for 14 frames to a

### Variable-Height Jumps

If the player releases the jump button while ascending, cut the upward velocity in half. A tap

```z80 id:ch19_variable_height_jumps
; check_jump_release -- Cut jump short if button released
;
; Input:  IX = player entity
; Output: dy halved if ascending and jump button not held
;
; Cost: ~40 T-states (Pentagon)

check_jump_release:
    ; Only relevant while ascending
```
```

```
    bit  7, (ix+10)       ; 20T  check sign of dy_int
    ret  z                ; 11/5T  not ascending (dy >= 0), skip

    ; Check if jump button is still held
    ; (assume A contains current input state from input handler)
    bit  4, a             ; 8T   bit 4 = fire/jump
    ret  nz               ; 11/5T  still held, do nothing

    ; Button released -- halve upward velocity
    ; Arithmetic right shift of 16-bit dy (preserves sign)
    ld   a, (ix+10)       ; 19T  dy_int
    sra  a                ; 8T   shift right arithmetic (sign-extending)
    ld   (ix+10), a       ; 19T
    ld   a, (ix+9)        ; 19T  dy_frac
    rra                   ; 4T   rotate right through carry (carry from SRA above)
    ld   (ix+9), a        ; 19T
    ret                    ; 10T
```

This is a 16-bit arithmetic right shift: `SRA` preserves the sign on the high byte, `RRA` picks up the carry on the low byte. Upward velocity halves, the arc flattens. Forty T-states for a vastly better-feeling jump.

Friction: Slowing Down on the Ground

When the player releases the direction keys, they should decelerate, not stop dead. The operation is a single right-shift of horizontal velocity.

“‘z80 id:ch19_friction_slowing_down_on_the ; apply_friction – Decelerate horizontal movement ; ; Input: IX = entity pointer ; Output: dx decayed toward zero ; ; Friction is applied as a right shift (divide by power of 2). ; SRA by 1 = divide by 2 (heavy friction, like rough ground) ; SRA by 1 every other frame = divide by ~1.4 (lighter friction) ; ; Cost: ~55 T-states (Pentagon)

apply_friction: ; Only apply friction on the ground bit 0, (ix+12) ; 20T on_ground flag ret z ; 11/5T in air – no ground friction

```
; 16-bit arithmetic right shift of dx (signed)
ld   a, (ix+8)        ; 19T  dx_int
sra  a                ; 8T   shift right, sign-extending
ld   (ix+8), a        ; 19T
ld   a, (ix+7)        ; 19T  dx_frac
rra                   ; 4T   rotate right through carry
ld   (ix+7), a        ; 19T
ret                    ; 10T


Shifting right by 1 divides velocity by 2 every frame -- the player stops within a few frames.

```z80 id:ch19_friction_slowing_down_on_the_2
; apply_friction_ice -- Light friction, every other frame
;
    ld   a, (frame_counter)
    and  1
    ret  nz               ; skip odd frames
```
```

```
    jr   apply_friction   ; apply on even frames only
```

Vary friction by surface type – look up the tile under the entity's feet and branch:

```
z80 id:ch19_friction_slowing_down_on_the_3     ; Determine surface type     ld
b, (ix+1)        ; player X     ld   a, (ix+3)        ; player Y     add  a,
(ix+14)       ; + height (feet position)     inc  a                ; one pixel
below feet     ld   c, a     call tile_at     cp   TILE_ICE     jr   z, .ice_friction
; Default: heavy friction     call apply_friction     jr   .done .ice_friction:
call apply_friction_ice .done:
```

Applying Velocity to Position

The final step: move the entity by its velocity via 16-bit fixed-point addition on each axis:

"'z80 id:ch19_applying_velocity_to_position ; move_entity – Apply velocity to position ; ; Input: IX = entity pointer ; Output: X and Y positions updated by dx and dy ; ; Cost: ~80 T-states (Pentagon)

move_entity: ; X position += dx (16-bit fixed-point add) ld a, (ix+0) ; 19T x_frac add a, (ix+7) ; 19T + dx_frac ld (ix+0), a ; 19T

```
ld   a, (ix+1)        ; 19T  x_int
adc  a, (ix+8)        ; 19T  + dx_int (with carry)
ld   (ix+1), a        ; 19T

; Y position += dy (16-bit fixed-point add)
ld   a, (ix+2)        ; 19T  y_frac
add  a, (ix+9)        ; 19T  + dy_frac
ld   (ix+2), a        ; 19T

ld   a, (ix+3)        ; 19T  y_int
adc  a, (ix+10)       ; 19T  + dy_int (with carry)
ld   (ix+3), a        ; 19T
ret                    ; 10T


### The Physics Loop

Putting it all together, the per-frame physics update for one entity looks like this:

```z80 id:ch19_the_physics_loop
; update_physics -- Full physics update for one entity
;
; Input:  IX = entity pointer
; Call order matters: gravity first, then move, then collide

update_entity_physics:
    call apply_gravity         ; accumulate downward velocity
    call apply_friction        ; decay horizontal velocity
    call move_entity           ; apply velocity to position
    call check_player_tiles    ; resolve tile collisions
    ret
```
```

The order is deliberate: forces first, then move, then collide. This is the standard for platformers. Total cost per entity: approximately 1,000-1,500 T-states (dominated by tile collision lookups at ~182T each). For 16 entities: 16,000-24,000 T-states, about 25-33% of the Pentagon frame budget. In practice, only the player and gravity-affected enemies need full tile collision checks – bullets and effects can use simpler bounds tests.

Part 3: Enemy AI

You do not have the T-states for pathfinding or decision trees. What you have is a jump table and a state byte. That is enough.

The Finite State Machine

Every enemy has a `state` byte (offset +5 in our entity structure) that selects which behaviour routine runs this frame:

State	Name	Behaviour
0	PATROL	Walk back and forth between two points
1	CHASE	Move toward the player
2	ATTACK	Fire a projectile or charge
3	RETREAT	Move away from the player
4	DEATH	Play death animation, then deactivate

Transitions are simple conditions: proximity checks, cooldown timers, health thresholds. Each is a comparison or bit test – never anything expensive.

The JP Table

The core of the AI dispatcher is a **jump table** indexed by the state byte. O(1) dispatch regardless of how many states you have:

“‘z80 id:ch19_the_jp_table ; ai_dispatch – Run the AI for one enemy entity ; ; Input: IX = enemy entity pointer ; Output: Entity state/position/velocity updated ; ; The state byte (ix+5) indexes into a table of handler addresses. ; Each handler is responsible for: ; 1. Executing this frame’s behaviour ; 2. Checking transition conditions ; 3. Setting (ix+5) to the new state if transitioning ; ; Cost: ~45 T-states dispatch overhead + handler cost

; State constants ST_PATROL equ 0 ST_CHASE equ 1 ST_ATTACK equ 2 ST_RETREAT equ 3 ST_DEATH equ 4

ai_dispatch: ld a, (ix+5) ; 19T load state byte add a, a ; 4T *2 (each table entry is 2 bytes) ld e, a ; 4T ld d, 0 ; 7T ld hl, ai_state_table ; 10T add hl, de ; 11T HL = table + state*2 ld e, (hl) ; 7T low byte of handler address inc hl ; 6T ld d, (hl) ; 7T high byte of handler address ex de, hl ; 4T HL = handler address jp (hl) ; 4T jump to handler ; (handler returns via RET)

ai_state_table: dw ai_patrol ; state 0 dw ai_chase ; state 1 dw ai_attack ; state 2 dw ai_retreat ; state 3 dw ai_death ; state 4

The `jp (hl)` instruction costs only 4 T-states -- the entire dispatch overhead is about 45 T-states regardless of state count. Note: `jp (hl)` jumps to the address *in* HL, not the addres

Patrol: The Dumb Walk

The simplest AI behaviour: walk in one direction until you reach a boundary, then turn around.

```z80 id:ch19_patrol_the_dumb_walk
; ai_patrol -- Walk back and forth between two points
;
; Input:  IX = enemy entity
; Output: Position updated, state may transition to CHASE
;
; The enemy walks at a constant speed (PATROL_SPEED).
; Direction is stored in bit 1 of the flags byte:
;   bit 1 = 0: moving right
;   bit 1 = 1: moving left
;
; Patrol boundaries are defined per-enemy type (hardcoded
; or stored in a level table). Here we use a simple range
; check against the initial spawn position +/- PATROL_RANGE.
;
; Cost: ~120 T-states (Pentagon)

PATROL_SPEED equ 1        ; 1 pixel per frame
PATROL_RANGE equ 32       ; 32 pixels from centre point

ai_patrol:
    ; --- Move in current direction ---
    bit  1, (ix+12)       ; 20T  check direction flag
    jr   nz, .move_left   ; 12/7T

.move_right:
    ld   a, (ix+1)        ; 19T  x_int
    add  a, PATROL_SPEED  ; 7T
    ld   (ix+1), a        ; 19T
    ; Check right boundary
    cp   PATROL_RIGHT_LIMIT ; 7T  (or use spawn_x + PATROL_RANGE)
    jr   c, .check_player ; 12/7T  not at edge yet
    ; Hit right edge -- turn left
    set  1, (ix+12)       ; 23T  set direction = left
    jr   .check_player    ; 12T

.move_left:
    ld   a, (ix+1)        ; 19T  x_int
    sub  PATROL_SPEED     ; 7T
    ld   (ix+1), a        ; 19T
    ; Check left boundary
    cp   PATROL_LEFT_LIMIT ; 7T
    jr   nc, .check_player ; 12/7T
```

```
    ; Hit left edge -- turn right
    res  1, (ix+12)       ; 23T

.check_player:
    ; --- Detection: is the player nearby? ---
    ; Simple range check: |player.x - enemy.x| < DETECT_RANGE
    ld   a, (player_x)    ; 13T  player x_int (cached in RAM)
    sub  (ix+1)           ; 19T  delta X
    jr   nc, .pos_dx      ; 12/7T
    neg                   ; 8T   absolute value
.pos_dx:
    cp   DETECT_RANGE     ; 7T   e.g., 48 pixels
    ret  nc               ; 11/5T  too far -- stay in PATROL

    ; Player detected -- transition to CHASE
    ld   (ix+5), ST_CHASE ; 19T  set state = CHASE
    ret                    ; 10T
```

A patrol enemy costs about 120 T-states per frame. That is trivial. Eight patrolling enemies cost under 1,000 T-states – barely a blip in the frame budget.

Chase: The Relentless Follower

The chase behaviour is simple: compute the sign of the horizontal distance between the enemy and the player, and move in that direction.

“‘z80 id:ch19_chase_the_relentless_follower ; ai_chase – Move toward the player ; ; Input: IX = enemy entity ; Output: Position updated, state may transition to ATTACK or RETREAT ; ; Cost: ~100 T-states (Pentagon)

CHASE_SPEED equ 2 ; faster than patrol

ai_chase: ; — Move toward player — ld a, (player_x) ; 13T sub (ix+1) ; 19T dx = player.x - enemy.x jr z, .vertical ; 12/7T same column – skip horizontal

```
; Sign of dx determines direction
jr   c, .chase_left   ; 12/7T  player is to the left (dx negative)
```

.chase_right: ld a, (ix+1) ; 19T add a, CHASE_SPEED ; 7T ld (ix+1), a ; 19T res 1, (ix+12) ; 23T face right jr .check_attack ; 12T

.chase_left: ld a, (ix+1) ; 19T sub CHASE_SPEED ; 7T ld (ix+1), a ; 19T set 1, (ix+12) ; 23T face left

.vertical: .check_attack: ; — Close enough to attack? — ld a, (player_x) sub (ix+1) jr nc, .pos_atk neg .pos_atk: cp ATTACK_RANGE ; 7T e.g., 16 pixels jr nc, .check_retreat ; 12/7T not close enough

```
; In attack range -- transition to ATTACK
ld   (ix+5), ST_ATTACK
ret
```

.check_retreat: ; — Low health? Retreat. — ld a, (ix+11) ; 19T health cp RETREAT_THRESHOLD ; 7T e.g., 2 out of 8 ret nc ; 11/5T health OK – stay in CHASE

```
; Health critical -- retreat
ld   (ix+5), ST_RETREAT
```

```
ret


The sign-of-dx technique: subtract, check carry. Carry set means player is to the left. Two in

### Attack: Fire and Cooldown

The ATTACK state fires a projectile, then waits for a cooldown timer. We reuse the `anim_frame

```z80 id:ch19_attack_fire_and_cooldown
; ai_attack -- Fire projectile, then cool down
;
; Input:  IX = enemy entity
; Output: May spawn a bullet, transitions back to CHASE when ready
;
; Cost: ~60 T-states (cooldown tick) or ~150 T-states (fire + spawn)

ATTACK_COOLDOWN equ 30    ; 30 frames between shots (0.6 seconds)

ai_attack:
    ; --- Cooldown timer ---
    ld   a, (ix+6)        ; 19T  anim_frame used as cooldown
    or   a                ; 4T
    jr   z, .fire         ; 12/7T  timer expired -- fire

    ; Decrement cooldown
    dec  (ix+6)           ; 23T
    ret                    ; 10T  wait

.fire:
    ; --- Spawn a bullet ---
    ; Find a free slot in the entity pool (bullet type)
    call find_free_entity  ; returns IY = free entity, or Z flag if none
    ret  z                 ; no free slots -- skip this shot

    ; Configure the bullet entity
    ld   a, (ix+1)
    ld   (iy+1), a         ; bullet X = enemy X
    ld   a, (ix+3)
    add  a, 4
    ld   (iy+3), a         ; bullet Y = enemy Y + 4 (mid-body)
    ld   (iy+4), TYPE_BULLET ; entity type
    ld   (iy+5), 0         ; state = 0 (active)

    ; Bullet direction: toward the player
    ld   a, (player_x)
    sub  (ix+1)
    jr   c, .bullet_left

.bullet_right:
    ld   (iy+8), BULLET_SPEED  ; dx_int = positive
    jr   .fire_done
```
```

```
.bullet_left:
    ld   a, 0
    sub  BULLET_SPEED
    ld   (iy+8), a         ; dx_int = negative (two's complement)

.fire_done:
    ; Set cooldown and return to CHASE
    ld   (ix+6), ATTACK_COOLDOWN  ; reset cooldown timer
    ld   (ix+5), ST_CHASE         ; back to chase state
    ret
```

The `find_free_entity` routine (from Chapter 18) scans for an inactive slot. If the pool is full, the shot is dropped.

Retreat: The Reverse Chase

The mirror of chase – compute sign of dx, move the other way:

“‘z80 id:ch19_retreat_the_reverse_chase ; ai_retreat – Move away from the player ; ; Input: IX = enemy entity ; Output: Position updated, transitions to PATROL if far enough away ; ; Cost: ~100 T-states (Pentagon)

RETREAT_DISTANCE equ 64 ; flee until 64 pixels away

ai_retreat: ; — Move away from player — ld a, (player_x) sub (ix+1) ; dx = player.x - enemy.x jr c, .flee_right ; player is left, so flee right

.flee_left: ld a, (ix+1) sub CHASE_SPEED ld (ix+1), a set 1, (ix+12) ; face left (fleeing) jr .check_safe

.flee_right: ld a, (ix+1) add a, CHASE_SPEED ld (ix+1), a res 1, (ix+12) ; face right (fleeing)

.check_safe: ; — Far enough away? Return to patrol — ld a, (player_x) sub (ix+1) jr nc, .pos_ret neg .pos_ret: cp RETREAT_DISTANCE ret c ; not far enough – keep fleeing

```
; Safe distance reached -- return to PATROL
ld   (ix+5), ST_PATROL
ret


### Death: Animate and Remove

Health reaches zero, state becomes DEATH. The handler plays an animation, then deactivates the

```z80 id:ch19_death_animate_and_remove
; ai_death -- Play death animation, then deactivate
;
; Input:  IX = enemy entity
; Output: Entity deactivated after animation completes
;
; Uses anim_frame as a countdown. When it reaches 0,
; the entity is marked inactive.
;
```
```

```
; Cost: ~40 T-states per frame

DEATH_FRAMES equ 8        ; 8 frames of death animation

ai_death:
    ld   a, (ix+6)        ; 19T  anim_frame (countdown)
    or   a                ; 4T
    jr   z, .deactivate   ; 12/7T

    dec  (ix+6)           ; 23T  count down
    ret                    ; 10T

.deactivate:
    res  7, (ix+12)       ; 23T  clear "active" flag (bit 7 of flags)
    ret                    ; 10T
```

Once bit 7 is cleared, the entity vanishes from rendering and its slot becomes available for reuse.

Optimisation: Update AI Every 2nd or 3rd Frame

Players cannot tell the difference between 50 Hz AI and 25 Hz AI. The screen and player input run at 50 fps, but enemy decisions at 25 fps (every 2nd frame) or 16.7 fps (every 3rd) are indistinguishable. Velocity carries the entity smoothly between AI ticks.

“‘z80 id:ch19_optimisation_update_ai_every ; update_all_ai – Update enemy AI on alternate frames ; ; Input: frame_counter = current frame number ; Output: All enemies updated (on even frames only)

update_all_ai: ld a, (frame_counter) ; 13T and 1 ; 7T check bit 0 ret nz ; 11/5T odd frame – skip AI entirely

```
; Even frame -- run AI for all active enemies
ld   ix, entity_array + ENTITY_SIZE  ; skip player (entity 0)
ld   b, MAX_ENEMIES    ; 8 enemies
```

.loop: push bc ; 11T save counter

```
; Check if entity is active
bit  7, (ix+12)        ; 20T
call nz, ai_dispatch   ; 17T + handler (only if active)

; Advance to next entity
ld   de, ENTITY_SIZE   ; 10T  16 bytes per entity
add  ix, de            ; 15T

pop  bc                ; 10T
djnz .loop             ; 13/8T
ret


This halves the AI cost. For 3rd-frame updating, use a modulo-3 check:

```z80 id:ch19_optimisation_update_ai_every_2
```
```

```
    ld   a, (frame_counter)
    ld   b, 3
    ; A mod 3: subtract 3 repeatedly
.mod3:
    sub  b
    jr   nc, .mod3
    add  a, b              ; restore: A = frame_counter mod 3
    or   a
    ret  nz                ; skip unless remainder is 0
```

The key insight: physics runs every frame for smooth movement. AI runs every 2nd or 3rd frame for decisions. The player sees fluid motion with slightly delayed reactions, and the result feels natural.

Part 4: Practical – Four Enemy Types

Four enemy types, each with distinct behaviour, wired into the entity system from Chapter 18.

1. The Walker – patrols a platform, reverses at edges. Detects the player by proximity. Chase behaviour: follow at ground level. Damage: contact only (no projectiles). Health: 1 hit.

State	Behaviour	Transition
PATROL	Walk left/right within range	Player within 48px: CHASE
CHASE	Move toward player at 2x speed	Within 16px: ATTACK
ATTACK	Pause, lunge forward	Cooldown expires: CHASE
DEATH	Flash 8 frames, deactivate	–

2. The Shooter – stands still (or patrols slowly), fires projectiles when the player is in range. Maintains distance.

State	Behaviour	Transition
PATROL	Slow walk or stationary	Player within 64px: ATTACK
ATTACK	Fire bullet, cooldown 30 frames	Player out of range: PATROL
RETREAT	Move away if player too close	Distance > 32px: ATTACK
DEATH	Explode animation, deactivate	–

3. The Swooper – moves vertically in a sine pattern (or simple up/down), dives toward the player when aligned.

“‘z80 id:ch19_part_4_practical_four_enemy ; ai_patrol_swooper – Vertical sine wave patrol ; ; Input: IX = swooper entity ; Output: Position updated with vertical oscillation ; ; Uses anim_frame as the sine table index, incrementing each AI tick ; ; Cost: ~80 T-states (Pentagon)

ai_patrol_swooper: ; Vertical oscillation ld a, (ix+6) ; 19T anim_frame = sine index inc (ix+6) ; 23T advance for next frame ld h, sine_table » 8 ; 7T sine table base (page-aligned, per Ch.4) ld l, a ; 4T index ld a, (hl) ; 7T signed sine value (-128..+127) sra a ; 8T /2 (reduce amplitude) sra a ; 8T /4 add a, (ix+3) ; 19T base Y + oscillation ld (ix+3), a ; 19T

```
; Check for dive: is player directly below?
ld   a, (player_x)
sub  (ix+1)
jr   nc, .pos
neg
```

.pos: cp 8 ; within 8 pixels horizontally? ret nc ; not aligned – stay patrolling

```
; Player below and aligned -- switch to dive (CHASE)
ld   (ix+5), ST_CHASE
ret


The Swooper uses the sine table from Chapter 4 for vertical oscillation. When the player passe

**4. The Ambusher** -- sits dormant until the player is very close, then activates aggressivel

```z80 id:ch19_part_4_practical_four_enemy_2
; ai_patrol_ambusher -- Dormant until player is adjacent
;
; Input:  IX = ambusher entity
; Output: Activates if player within 16 pixels
;
; Cost: ~50 T-states (Pentagon)

AMBUSH_RANGE equ 16

ai_patrol_ambusher:
    ; Check proximity (Manhattan distance for cheapness)
    ld   a, (player_x)
    sub  (ix+1)
    jr   nc, .px
    neg
.px:
    ld   b, a              ; |dx|

    ld   a, (player_y)
    sub  (ix+3)
    jr   nc, .py
    neg
.py:
    add  a, b              ; Manhattan distance = |dx| + |dy|
    cp   AMBUSH_RANGE
    ret  nc                ; too far -- stay dormant

    ; Player is close -- activate!
    ld   (ix+5), ST_CHASE  ; go straight to aggressive chase
```
```

```
    ; (Could also play an activation sound/animation here)
    ret
```

Manhattan distance (|dx| + |dy|) costs about 30 T-states versus ~200 for Euclidean. For proximity checks, it is good enough.

Wiring It Into the Game Loop

The complete per-frame update, building on Chapter 18:

"'z80 id:ch19_wiring_it_into_the_game_loop game_frame: halt ; wait for VBlank

```
; --- Input ---
call read_input            ; Chapter 18

; --- Player physics ---
ld   ix, entity_array      ; player is entity 0
call handle_player_input   ; set dx from keys, try_jump from fire
call update_entity_physics ; gravity + friction + move + tile collide

; --- Enemy AI (every 2nd frame) ---
call update_all_ai

; --- Enemy physics (every frame) ---
call update_all_enemy_physics

; --- Entity-vs-entity collisions ---
call check_all_collisions

; --- Render ---
call render_entities       ; Chapter 16 sprites
call update_music          ; Chapter 11 AY

jr   game_frame


The `check_all_collisions` routine tests player vs enemies and bullets vs entities:

```z80 id:ch19_wiring_it_into_the_game_loop_2
; check_all_collisions -- Test player vs enemies, bullets vs enemies
;
; Cost: ~2,000-3,000 T-states depending on active entity count

check_all_collisions:
    ld   ix, entity_array       ; player entity
    ld   iy, entity_array + ENTITY_SIZE
    ld   b, MAX_ENEMIES + MAX_BULLETS  ; 8 enemies + 7 bullets

.loop:
    push bc

    ; Skip inactive entities
    bit  7, (iy+12)
```
```

```
    jr   z, .next

    ; Is this an enemy? Check player vs enemy
    ld   a, (iy+4)              ; entity type
    cp   TYPE_BULLET
    jr   z, .check_bullet

    ; Enemy: test against player
    call check_aabb
    jr   nc, .next              ; no collision
    call handle_player_hit      ; damage player, knockback, etc.
    jr   .next

.check_bullet:
    ; Bullet: check against all enemies (or just nearby ones)
    ; For simplicity, check bullet source -- don't hit the shooter
    ; This is handled by a "source" field or by checking type
    call check_bullet_collisions

.next:
    ld   de, ENTITY_SIZE
    add  iy, de
    pop  bc
    djnz .loop
    ret
```

Agon Light 2 Notes

The same physics and AI code runs unchanged on the Agon – pure Z80 arithmetic with no hardware dependencies. The Agon's ~368,000 T-state budget means you can afford more entities (32 or 64), per-frame AI (no 2nd-frame skip needed), more collision check points, and richer state machines. Keep the physics constants identical between platforms so the game *feels* the same. The Agon VDP provides hardware sprite collision for bullet-vs-enemy checks, but tile collision remains a Z80 tilemap lookup.

Tuning Guide

The numbers in this chapter are starting points, not commandments. Here is a reference table for tuning the feel of your platformer:

Parameter	Value	Effect
GRAVITY_FRAC	$20 (0.125)	Floaty, moon-like
GRAVITY_FRAC	$40 (0.25)	Standard platformer feel
GRAVITY_FRAC	$60 (0.375)	Heavy, fast-falling
JUMP_INT	$FD (-3)	Low jump (~2 tiles)
JUMP_INT:FRAC	$FC:$80 (-3.5)	Medium jump (~3 tiles)
JUMP_INT	$FB (-5)	High jump (~5 tiles)

Parameter	Value	Effect
PATROL_SPEED	1	Slow, predictable
CHASE_SPEED	2	Matches player walk speed
CHASE_SPEED	3	Faster than player – forces jumping
DETECT_RANGE	32	Short range, enemy is “dumb”
DETECT_RANGE	64	Medium range, balanced
DETECT_RANGE	128	Long range, enemy is aggressive
ATTACK_COOLDOWN	15	Rapid fire (2 shots/second at 25 Hz AI)
ATTACK_COOLDOWN	30	Moderate fire rate
ATTACK_COOLDOWN	60	Slow, deliberate
Friction shift	»1 every frame	Stops in ~3 frames (sticky)
Friction shift	»1 every 2 frames	Stops in ~6 frames (smooth)
Friction shift	»1 every 4 frames	Stops in ~12 frames (ice)

Play-test constantly. Change one number, play for thirty seconds, feel the difference. Physics tuning is not engineering – it is craft. The numbers should be in a constants block at the top of your source file, clearly labelled, easy to modify.

Summary

- **AABB collision** uses four comparisons with early exit. Most pairs are rejected after one or two tests. Cost: 91-270 T-states per pair on the Z80 (IX/IY indexed addressing dominates). Order the tests to reject the most common non-collision case first (usually horizontal). Watch for 8-bit overflow when computing `x + width` near screen edges.
- **Tile collision** converts pixel coordinates to a tile index via right-shift and lookup. O(1) per point checked, regardless of map size. Check the four corners and edge midpoints of the entity’s bounding box.
- **Sliding collision response** resolves collisions on each axis independently. Apply X velocity then check X collisions; apply Y velocity then check Y collisions. Diagonal motion against a wall naturally becomes sliding.
- **Gravity** is a fixed-point addition to vertical velocity every frame: `dy += gravity`. With 8.8 format, sub-pixel values like 0.25 pixels/frame^2 produce smooth, natural-feeling acceleration curves.
- **Jumping** sets vertical velocity to a negative value. Gravity decelerates it, producing a parabolic arc with no explicit curve calculation. Variable-height jumps cut the velocity in half when the button is released.
- **Friction** is a right-shift of horizontal velocity: `dx >>= 1`. Vary the frequency of application for different surface types (every frame = rough ground, every 4th frame = ice).
- **Enemy AI** uses a finite state machine with JP-table dispatch. Five states (Patrol, Chase, Attack, Retreat, Death) cover most platformer enemy behaviours. Dispatch cost: ~45 T-states regardless of state count.
- **Chase** uses the sign of `player.x - enemy.x` for direction. Two instructions, zero trigonometry.
- **Update AI every 2nd or 3rd frame** to halve or third the CPU cost. Physics runs every frame for smooth movement; AI decisions can lag by 1-2 frames without the player noticing.

- **Four enemy types** (Walker, Shooter, Swooper, Ambusher) demonstrate how the same state machine framework produces varied behaviours by changing a few constants and one or two state handlers.
- **Total cost** for a 16-entity game (physics + collisions + AI): approximately 15,000-20,000 T-states per frame on the Spectrum (about 25-28% of the Pentagon budget), leaving room for rendering and sound.

Try It Yourself

1. **Build the AABB test.** Place two entities on screen. Move one with the keyboard. Change the border colour when they collide. Verify the early-exit behaviour by placing entities far apart and measuring T-states with the border-colour harness from Chapter 1.

2. **Implement tile collision.** Create a simple tilemap with solid blocks and empty space. Move the player with keyboard input and gravity. Verify that the player lands on platforms, cannot walk through walls, and slides along surfaces when moving diagonally.

3. **Tune the physics.** Using the tuning guide above, adjust gravity and jump force to create three different feels: floaty (moon), standard (Mario-like), and heavy (Castlevania-like). Play each for a minute and note how the constants change the experience.

4. **Build all four enemy types.** Start with the Walker (patrol + chase only), then add the Shooter (projectiles), the Swooper (sine-wave motion), and the Ambusher (dormant activation). Test each one individually before combining them in one level.

5. **Profile the frame budget.** With all 16 entities active, use the multi-colour border profiler (Chapter 1) to visualise how much of the frame is spent on physics (red), AI (blue), collisions (green), and rendering (yellow). Adjust the AI update frequency and measure the difference.

Sources: Dark "Programming Algorithms" (Spectrum Expert #01, 1997) – fixed-point arithmetic foundations; Game development folklore and Z80 platform knowledge; Jump table technique is standard Z80 practice documented across the ZX Spectrum development community

Chapter 20: Demo Workflow — From Idea to Compo

> *"Design is the complete aggregate of all demo components, both visible and concealed. Design characterises realizational, stylistic, ideological integrity."* – Introspec, "For Design," Hype, 2015

A demo is not built in a single session of inspired coding. It is a project – one with deadlines, dependencies, creative decisions that must be locked down weeks before the party, technical gambles that pay off or do not, and a final submission that either works on the compo machine or fails in front of an audience. The distance between "I have an idea for a demo" and "it placed third at DiHalt" is measured not in lines of code but in workflow: how you organise effects, how you schedule your time, how you build and test, how you handle the inevitable moment when the music is not ready and the party is in four days.

This chapter is about that workflow. We have spent nineteen chapters on techniques – inner loops, cycle counting, compression, sound, sync. Now we step back and ask: how does a real demo come together? How do you go from a blank screen to a two-minute production that runs reliably, looks intentional, and lands at the right party on the right day?

The answers come from three sources. restorer's making-of article for Lo-Fi Motion (Hype, 2020) provides a detailed case study of a working production pipeline – fourteen effects built in two weeks of evening coding, with a scene table system and a toolchain that any reader can replicate. Introspec's philosophical essays on Hype – "For Design" and "MORE" (both 2015) – articulate the design thinking that separates a collection of effects from a coherent demo. And the broader making-of culture of the ZX Spectrum scene – from Eager's detailed NFO to GABBA's iOS video editor workflow to NHBF's 256-byte puzzle – gives us a gallery of approaches to learn from.

20.1 What "Design" Means in a Demo

When demosceners say "design," they do not mean graphic design. They do not mean UI layout or colour theory, although those matter. Introspec's definition, published on Hype in January 2015, is broader and more demanding:

> Design is the complete aggregate of all demo components, both visible and concealed.

This definition includes the effects the audience sees, the transitions between them, the music choice and how it synchronises with visuals, the colour palette, the pacing, the emotional arc – but also the code architecture, the memory layout, the compression strategy, the build pipeline, and the decisions about what to leave out. A demo with beautiful pixel art and terrible pacing has poor design. A demo with crude visuals but perfect musical sync and a clear emotional arc might have excellent design. A deliberately ugly demo, one that chooses its aesthetic with intention, can have outstanding design.

The implications for workflow are immediate. If design encompasses everything, then design decisions happen at every stage. The choice of assembler constrains what your build pipeline can do. The memory map determines which effects can coexist. The order in which you build effects determines what you can cut if time runs out. Every technical choice has an aesthetic consequence, and every aesthetic choice has a technical cost.

Demo production must be simultaneously bottom-up and top-down, with constant feedback between creative vision and technical reality. The workflow must support that feedback loop.

20.2 Lo-Fi Motion: A Complete Case Study

In September 2020, restorer published a making-of article for Lo-Fi Motion, a ZX Spectrum demo released at DiHalt 2020. The article is valuable not for any single technical insight but because it documents an *entire production pipeline* – from initial concept to finished binary – in enough detail to reproduce.

The Concept: "Belarusian Pixel"

Lo-Fi Motion uses attribute-resolution graphics – what restorer calls "lo-fi" rendering. Most effects work at the 32x24 attribute grid or at a doubled 32x48 resolution using half-character rows. No pixel-level rendering. The aesthetic is deliberately blocky, embracing the ZX Spectrum's attribute grid rather than fighting it. The name says it: this is lo-fi, and the motion is the point.

This is a design decision with cascading technical benefits. Attribute-resolution effects are cheap to calculate (192 or 384 bytes per frame instead of 6,144), cheap to store (small frame buffers mean more room for compressed data), and fast to display (writing 768 bytes to attribute RAM fits easily within one frame). The lo-fi aesthetic is not a compromise – it is a choice that unlocks fourteen effects in two weeks of evening work.

The Scene Table

At the centre of Lo-Fi Motion's architecture is the **scene table** – a data structure that drives the entire demo. Each entry in the table describes one scene:

```
z80 id:ch20_the_scene_table ; Scene table entry (conceptual structure) scene_entry:
DB  bank_number          ; which 16K memory bank holds this effect's code     DW
entry_address        ; start address of the effect routine     DW  frame_duration
; how many frames this scene runs     DB  param_byte_1         ; effect-specific
```

```
parameter     DB  param_byte_2         ; effect-specific parameter     ; ...
additional parameters as needed
```

The demo engine reads the scene table linearly. For each entry, it pages in the specified memory bank, jumps to the entry address, and runs the effect for the specified number of frames. When the duration expires, it advances to the next entry. The entire demo – all fourteen effects, all transitions, all timing – is encoded in this one table.

This is the same architectural pattern we saw in Chapter 12's scripting engine, stripped to its essentials. The Eager scripting engine had two levels (outer script for effects, inner script for parameter variations) and the kWORK command for asynchronous frame generation. Lo-Fi Motion's scene table is simpler: one level, synchronous generation, no asynchronous buffering. The simplicity is the point. It works. It was built in two weeks.

The scene table pattern has a critical workflow advantage: it separates content from engine. Adding a new effect means writing the effect routine and adding one entry to the table. Reordering the demo means rearranging table entries. Adjusting timing means changing duration values. The engine code does not change. This separation means you can iterate on the demo's structure – its pacing, its order, its timing – without touching the engine, and iterate on individual effects without touching the structure.

Fourteen Effects

Lo-Fi Motion contains approximately fourteen distinct visual effects. restorer lists them by their working names: raskolbas, slime, fire, interp, plasma, rain, dina, rtzoomer, rbars, bigpic, and several others. Each effect is a self-contained routine that renders into a virtual buffer.

The virtual buffer is a key architectural choice. Most effects do not write directly to screen memory. Instead, they render to a **1-byte-per-pixel buffer** – a block of RAM where each byte represents one attribute cell's colour value. The buffer is typically 32 bytes wide and 24 or 48 bytes tall (for half-character resolution). After the effect renders into the buffer, a separate output routine copies the buffer to attribute RAM, performing any necessary format conversion.

This indirection costs a few hundred T-states per frame but provides two benefits. First, effects are isolated from the screen's physical layout. An effect that renders into a linear buffer does not need to know about the attribute memory's address structure. Second, effects can be composed: two effects can render into separate buffers, and a mixing routine can blend them before output. Lo-Fi Motion uses this for transitions – crossfading between two effects by interpolating their buffer values.

The buffer also enables the half-character resolution mode. A 32x48 buffer maps to the screen by using two attribute writes per character cell (one for the "top half" and one for the "bottom half"), exploiting the timing trick of rewriting attributes mid-scanline. This doubles the vertical resolution at the cost of more complex output code and tighter timing constraints.

The Effects Themselves

Each effect is a variation on themes from earlier chapters: **plasma** (sum-of-sines from Chapter 9), **rotozoomer** (texture-walking from Chapter 7), **fire** (cellular automaton averaging neighbours), **rain** (particle system), and **bigpic** (pre-compressed bitmap animation decompressed frame-by-frame, using techniques from Chapter 14). None of these effects is novel. The point is that at attribute resolution, every one of them is cheap enough that fourteen fit in one demo built in two weeks. The lo-fi decision is a force multiplier.

The Toolchain

restorer's toolchain for Lo-Fi Motion is a concrete answer to the question "what tools do I need to make a demo?"

Assembler: sjasmplus. The standard Z80 macro assembler for the modern ZX scene. Memory banking (SLOT/PAGE directives), conditional assembly, macros, INCBIN for embedded data, DISPLAY for build-time diagnostics, output to .tap/.sna/.trd. The scene table, effect code, compressed data, and engine all compile in one sjasmplus invocation.

Emulator: zemu. restorer's emulator of choice for Lo-Fi Motion. Unreal Speccy and Fuse are equally common. What matters is accurate timing and fast reload – you need to test a new build every few minutes.

Graphics: BGE 3.05 + Photoshop. BGE (Burial Graphics Editor, by Sinn/Delirium Tremens) is a ZX Spectrum-native graphics editor, widely used in the Russian scene for creating attribute-level artwork directly on the target platform. Pre-rendered PC images go through Photoshop (or Multipaint, GIMP) and custom scripts.

Scripts: Ruby. Conversion pipeline automation: images to attribute data, sine tables to binary includes, animation sequences to delta-compressed streams. Python, Perl, and Processing are equally common. What matters is that conversion is automated and repeatable.

Compression: hrust1opt. Hrust 1 with optimal parsing. The Z80 decompressor is relocatable (uses the stack for its working buffer), convenient for demos that page data in and out of banked memory.

The practical lesson: there is no single "right" toolchain. The right one is where every step from source asset to final binary is automated, changing one input regenerates all dependent outputs, and the whole build completes in seconds. Any manual step is a bug waiting to happen at 2 AM before the compo deadline.

The Build Pipeline

The tools chain together through a **Makefile** (or equivalent build script). The pipeline for Lo-Fi Motion looks approximately like this:

```
Source assets (PNG, raw data)
    |
    v
Ruby conversion scripts
    |
    v
```

```
Binary includes (.bin, .hru)
    |
    v
sjasmplus assembly
    |
    v
Output binary (.trd or .sna)
    |
    v
Test in emulator (zemu)
```

Each arrow is a Makefile rule. Change a PNG, run `make`, and the entire chain re-executes – conversion, compression, assembly – producing a fresh binary in seconds. Load it in the emulator, watch the result, decide what to change, edit the source, run `make` again. This edit-build-test loop, measured in seconds, is what makes it possible to build fourteen effects in two weeks.

The Makefile also serves as documentation. Reading the build rules tells you exactly which scripts produce which outputs, which effects depend on which data files, and what the complete dependency graph looks like. When you return to the project after a six-month break, the Makefile tells you how everything fits together.

The Timeline: Two Weeks of Evenings

Lo-Fi Motion was built in approximately two weeks of evening coding sessions. restorer was working a day job. The evenings were the only available time.

This timeline is realistic for a lo-fi attribute demo, and instructive for anyone planning their first production. The breakdown looks roughly like this:

- **Days 1-2:** Engine architecture. Scene table system, virtual buffer, output routine, basic framework. Get one effect (plasma) running through the full pipeline.
- **Days 3-7:** Effects. Two to three effects per evening once the framework is solid. Each effect is 100-300 lines of assembly, rendering into the virtual buffer. Test each one individually.
- **Days 8-10:** Content. Pre-rendered images, font data, conversion scripts. This is where the Ruby scripts earn their keep.
- **Days 11-12:** Integration. All effects in the scene table, timing adjusted to the music, transitions tuned. This is where the scene table's edit-and-rebuild workflow pays off.
- **Days 13-14:** Polish and debugging. Border colours for timing visualisation (Chapter 1), fixing effects that break on edge cases, final compression pass to fit everything in memory.

The critical observation: the engine and pipeline consume the first two days. Every subsequent day benefits from that investment. If you skip the pipeline work and hardcode your first effect directly into screen memory, you save a day up front and lose a week later when you try to add a second effect and discover that nothing is modular.

20.3 Making-of Culture

The ZX Spectrum demoscene has a strong culture of documenting how demos are made. This is not universal in the broader demoscene – on many platforms, demos ship with no documentation beyond credits. On the Spectrum scene, detailed making-of articles are a tradition, and Hype (hype.retroscene.org) is the primary venue for publishing them.

Eager: The Technical NFO

When Introspec released Eager (to live) at 3BM Open Air 2015, the ZIP file included a file_id.diz – the traditional demo information file – that went far beyond credits and greetings. It was a technical writeup: the attribute tunnel approach, the four-fold symmetry optimisation, the digital drum hybrid technique, the asynchronous frame generation architecture. Kylearan, reviewing the demo on Pouet, wrote: "Big thanks for the nfo file alone, I love reading technical write-ups! Helps in understanding what I'm seeing/hearing, too."

Introspec then published an even more detailed making-of article on Hype, which became the primary source for Chapters 9 and 12 of this book. The article explained not just *what* the demo does but *why* – the reasoning behind each technical decision, the constraints that drove the architecture, the creative goals that shaped the visual design.

This level of documentation serves multiple purposes. For the audience, it deepens appreciation – understanding how an effect works makes watching it more rewarding, not less. For other coders, it is education – the making-of articles on Hype are the closest thing the ZX scene has to a technical curriculum. For the author, it is a form of closure – articulating the decisions forces you to understand your own work, and the community feedback (Hype comments can be hundreds of posts deep) stress-tests your reasoning.

GABBA: A Different Workflow

diver4d's making-of article for GABBA (2019) documents a radically different workflow from Eager's. Where Introspec spent weeks on a scripting engine and asynchronous frame buffer, diver4d used Luma Fusion – an iOS video editor – as his synchronisation tool.

We covered the technical details in Chapter 12. The workflow insight is what matters here: diver4d recognised that frame-level audio-visual synchronisation is a *video editing* problem, not a *programming* problem. By doing the sync work in a tool designed for it, he could iterate on timing in seconds instead of minutes. The Z80 code was the implementation layer; the creative decisions happened in the video editor.

This is a general principle. Demo workflow is not about doing everything in assembly. It is about using the right tool for each task. Assembly for inner loops. Processing or Ruby for code generation. Photoshop or Multipaint for graphics. A video editor for timing. A Makefile to tie them together. The demo is the output; the tools are whatever gets you there fastest.

NHBF: The Puzzle

UriS's making-of for NHBF (2025) documents a workflow at the opposite extreme from Lo-Fi Motion's fourteen-effect pipeline. NHBF is a 256-byte intro – the entire program, code and data, fits in less space than a single attribute frame. The "workflow" is one person staring at a hex dump, constantly reshuffling instructions to find shorter encodings, discovering that register values from one routine happen to match the data needs of another.

We covered the specific techniques in Chapter 13. The workflow lesson is about constraint-driven creativity. UriS describes the process as "playing puzzle-like games" – an apt metaphor because the optimisation space in 256-byte coding is combinatorial. You cannot plan a path to the solution. You can only keep rearranging pieces and remain alert for serendipitous alignments. Art-Top's discovery that register values from the screen-clearing routine matched the text string length was not planned. It was noticed.

This matters for demo workflow at any scale. Even in a full-sized demo with a proper engine and a Makefile and a scene table, there are moments when the best solution comes from stepping back and looking at the whole picture, noticing an accidental alignment between two systems that were designed independently. The puzzle-solving mindset is not exclusive to size-coding. It is a mode of thinking that improves all demo work.

20.4 The Toolchain in Detail

The ZX Spectrum demo toolchain has converged on a standard set. Here is a typical project layout:

```
src/
    main.asm            ; entry point, scene table, engine loop
    engine.asm          ; scene table interpreter, buffer management
    effects/
        plasma.asm      ; individual effect routines
        fire.asm
        rotozoomer.asm
    sound/
        player.asm      ; music player (PT3 or custom)
        drums.asm       ; digital drum sample playback
    data/
        music.pt3       ; music file (INCBIN)
        screens.zx0     ; compressed graphics (INCBIN)
        sinetable.bin   ; pre-generated lookup table (INCBIN)
Makefile
tools/
    gen_sinetable.rb    ; Ruby script: generate sine table
    convert_gfx.rb      ; Ruby script: PNG to attribute data
```

Assembler: sjasmplus

The workhorse. Memory banking via SLOT/PAGE directives, conditional assembly, macros, INCBIN for embedded data, DISPLAY for build-time diagnostics, and output to .tap/.sna/.trd. A typical demo compiles in a single sjasmplus invocation.

Emulators

Unreal Speccy is preferred by many Russian-scene demosceners for its deterministic timing and accurate Pentagon emulation, with TR-DOS, TurboSound, and multiple clone model support. **Fuse** is widely available on Linux and macOS. **zemu** is another option, used by restorer for Lo-Fi Motion. For source-level debugging, **DeZog** in VS Code connects to ZEsarUX and provides breakpoints, register inspection, and memory views.

Pick one emulator for primary development. Test on others before release. Demos that work in one emulator and crash in another are a party tradition best avoided.

Graphics and Code Generation

Multipaint enforces attribute constraints in real time – purpose-built for 8-bit pixel art. **Photoshop, GIMP, or Aseprite** offer creative freedom but require conversion scripts (Python, Ruby, Processing) to quantise and export. **Processing** handles generative graphics and code generation – Introspec used it to generate the chaos zoomer's unrolled code sequences (Chapter 9).

Build Automation and CI

Your Makefile must automate the full pipeline: source assets to conversion scripts to compression to assembly. If any step requires manual intervention, it will fail at 2 AM before the deadline.

CI via GitHub Actions is increasingly common. A workflow that builds on every push catches implicit dependencies – the demo assembles on your machine but fails on a clean environment because of an undeclared tool version. Lo-Fi Motion's source is on GitHub, published as a reference implementation: clone it, run `make`, get a working binary. This openness is unusual in the demoscene and valuable for learning.

20.5 Compo Culture

A demo without a compo is a video on YouTube. A demo at a compo is a performance – shown on a big screen, with an audience, with other entries to compare against, with prizes at stake. The compo is where the work meets its audience, and the culture around compos shapes the work.

The Major Parties

The ZX Spectrum demo scene is served by a handful of recurring parties, each with its own character.

Chaos Constructions (CC) is the largest and most prestigious ZX demo event, held in Saint Petersburg, Russia. The ZX demo compo at CC draws the strongest entries: Break Space (2016), Eager's successors, and productions from groups like Thesuper, 4th Dimension, and Placeholders. CC is where you go to compete at the highest level. The audience is large, knowledgeable, and unforgiving.

DiHalt is held in Nizhny Novgorod, Russia, and has both a summer event and a "Lite" winter edition. DiHalt tends to be more experimental than CC – the audience is welcoming to first-time entrants, and the atmosphere encourages risk-taking. Lo-Fi Motion was released at DiHalt 2020. If you are entering your first compo, DiHalt Lite is a good choice.

Multimatograf is a smaller event with a tradition of encouraging new work. The compo categories are broad, the entry requirements are minimal, and the vibe is supportive. Introspec has reviewed Multimatograf compos on Hype, sometimes critically – he holds every party to the same standard – but the event itself is welcoming to beginners.

CAFe (Creative Art Festival) is a demoscene event with a broader scope (not exclusively ZX), but the ZX categories attract strong entries. GABBA took first place at CAFe 2019.

Revision is the world's largest demoscene event, held annually in Saarbrucken, Germany. It is not ZX-specific, but the "8-bit demo" and "oldschool" categories welcome ZX Spectrum entries. Competing at Revision means showing your work to the global demoscene – an audience of thousands, most of whom have never seen a Spectrum demo. SerzhSoft's Megademica won the 4K intro compo at Revision 2019, proving that ZX entries can compete on the global stage.

How to Enter Your First Compo

The process is less intimidating than it sounds.

1. Choose a party. Start with a smaller event – DiHalt Lite, Multimatograf, or a local party if one exists in your area. Larger parties have higher expectations, and the pressure of competing against experienced groups at CC can be counterproductive for a first entry.

2. Know the rules. Each party publishes compo rules specifying: platform requirements (which Spectrum model, which emulator configuration), file format (.tap, .trd, .sna), maximum file size, whether remote entries are accepted, and submission deadlines. Read the rules. Follow the rules. A technically impressive demo that ships as a .tzx when the rules require .trd will be disqualified.

3. Test on the target platform. If the party runs entries on real hardware (a physical Pentagon or Scorpion), test on that hardware or on an emulator configured to match. Demos that work perfectly on one machine model and crash on another are distressingly common. The differences are subtle: contended memory timings, bank switching delays, AY chip quirks. Chapter 15 covers the machine-specific details; Chapter 5 of Introspec's GO WEST series covers the portability pitfalls.

4. Submit early. Most parties accept remote entries via email or a web form. Submit a day early if possible. Last-minute submissions are stressful and prone to errors (uploading the wrong file, forgetting to include a required metadata file).

The party version can be imperfect – many demos are updated to "final" versions after the party, fixing bugs discovered during the compo showing.

5. Write a file_id.diz or NFO. Include a text file with credits (who did what), platform requirements (which model, which mode), and – if you are willing – a brief technical description. The audience appreciates knowing what they are looking at. The scene appreciates the documentation. And you will appreciate having written it when, three years later, you try to remember how the plasma table generation works.

6. Watch the compo. If you are at the party in person, watch your demo on the big screen with the audience. The experience of seeing your work displayed publicly, of hearing the audience react, of comparing your entry to the others – this is why compos exist. If you are submitting remotely, watch the stream if one is available. Some parties publish compo recordings on YouTube afterward.

7. Do not expect to win. Your first entry is a learning experience. The goal is to finish something, submit it, and see it shown. Placing is a bonus. The feedback you get – from the audience, from other sceners, from your own reaction to seeing it on a big screen – is worth more than any prize.

Remote entries are accepted at most ZX events. Lo-Fi Motion was a remote entry at DiHalt 2020. Some parties run online-only events streamed on YouTube or Twitch. If your nearest demoscene event is a 12-hour flight away, online compos are a viable starting point.

20.6 The Community

The ZX Spectrum demoscene is small enough that most active participants know each other, and large enough that it sustains multiple active communities.

Hype (hype.retroscene.org)

The primary Russian-language forum for ZX Spectrum demoscene discussion. Founded and moderated by Introspec, it hosts the making-of articles, technical tutorials, compo reviews, and design discussions that form the core source material for this book. Threads run to hundreds of comments, with experienced coders debating cycle counts in detail. For a non-Russian speaker, browser translation tools handle the prose well enough, and Z80 assembly reads the same in every alphabet.

The culture is direct. If you post a demo with a timing bug, someone will tell you exactly which T-state is wrong. This directness produces genuine technical discussion rather than polite but unhelpful encouragement.

ZXArt (zxart.ee)

The comprehensive archive of ZX Spectrum creative works – demos, music, graphics, games, magazines, and metadata. Every production in this book can be found on ZXArt with screenshots, credits, party results, and downloads. ZXArt also hosts digitised ZX magazines in TRD format (Spectrum Expert, Born Dead, ZX Format), containing the original articles that established the techniques this book teaches.

Pouet (pouet.net)

The global demoscene production database. For the ZX scene, Pouet bridges to the wider community – ZX demos rated by people who primarily watch PC or Amiga productions. The perspective shift is valuable: a technically brilliant inner loop that impresses Hype readers might be invisible to a Pouet commenter who focuses on visual impact and music sync. Pouet also hosts NFO files – when you cannot find a making-of article on Hype, check the NFO on Pouet.

20.7 Project Management for Demo Makers

Demo making is project management. The project has a deadline (the party date), deliverables (the final binary), dependencies (music, graphics, effects, engine), and usually a team of contributors with competing priorities. Managing this is not glamorous, but it is what separates finished demos from abandoned prototypes.

The Minimum Viable Demo

Start with the simplest possible version of your demo that is complete – not polished, not impressive, but complete. One effect, one piece of music, a proper beginning and end. Get this running end-to-end through the full build pipeline within the first few days. This is your safety net. If everything goes wrong – if the complex effect you planned does not work, if the musician is late with the final track, if your hard drive dies a week before the party – you have something to submit.

Then iterate. Add effects one at a time. Replace the placeholder music when the final track arrives. Add transitions, polish timing, optimise memory usage. Each iteration produces a complete, submittable demo that is better than the last. At any point, you can stop and submit what you have.

This incremental approach is how Lo-Fi Motion was built. restorer did not write fourteen effects and then stitch them together. He built the engine and one effect, verified they worked, then added effects one by one. Each evening's work produced a slightly better demo. If he had run out of time at ten effects instead of fourteen, the demo would still have been complete and submittable.

Working with Collaborators

Most demos are collaborations. Three principles keep them on track:

Establish the data format early. The musician needs to know: PT3 or custom player? Single AY or TurboSound? How are drum triggers signalled? The artist needs to know: attribute resolution or pixel resolution? Colour constraints? Maximum file size? Receiving a TurboSound composition when your engine supports only single AY is a disaster, and it is your fault for not specifying constraints.

Communicate the timeline. If the party is in four weeks, tell the musician you need the track in two. The buffer is for integration, debugging, and surprises.

Provide placeholders. Use a placeholder .pt3 with the right tempo until the final track arrives. Use programmer art until the final graphics arrive. The engine

should never depend on final assets. When real assets arrive, drop them into the pipeline and rebuild.

Debugging and Testing

Demo bugs are uniquely painful because they manifest in front of an audience. A crash during the compo showing is both a technical failure and a social embarrassment. Testing is not optional.

Test on multiple emulators. Each emulator has slightly different timing, memory initialisation, and AY behaviour. A demo that works in Unreal Speccy but crashes in Fuse probably has a timing or memory assumption that is valid on Pentagon but not on standard Spectrum.

Test from a cold start. Clear all memory before loading the demo. Do not assume any register values or memory contents from a previous program. If your demo works after running a previous demo but crashes from a fresh boot, you have an initialisation bug.

Test the compo file, not the development binary. The file you submit should be the exact file you tested. Not a "quickly recompiled" version with a last-minute fix. Last-minute fixes introduce last-minute bugs.

Use border colours for timing. The technique from Chapter 1: set the border to different colours at different points in the frame loop. If the border flash extends into the visible area, your code is too slow. If it does not, you have margin. This is the fastest way to verify that an effect fits within the frame budget.

20.8 Transcending the Platform: Introspec's "MORE"

In February 2015, Introspec published a short essay on Hype titled simply "MORE." It is not a technical article. It contains no code, no cycle counts, no inner loops. It is a challenge to the ZX Spectrum scene – and by extension, to everyone who works within hardware constraints.

The argument is that the best demos are not the ones that do the most impressive things *despite* the platform limitations. They are the ones that transcend the platform entirely – that create experiences which would be meaningful on any hardware. The platform's constraints shape the technique but should not limit the ambition.

> Two pixels suffice to tell a story.

This is Introspec's most quoted line. It means: the artistic content of a demo is not determined by its resolution, its colour depth, its polygon count, or its sample rate. Two pixels – two attribute cells, two dots on a 32x24 grid – can tell a story if the timing is right, the context is clear, and the intention is genuine. The technology serves the art, not the other way around.

Introspec references James Houston's "Big Ideas (Don't Get Any)" – a video where a Sinclair ZX Spectrum, dot matrix printer, and other obsolete hardware perform a Radiohead song. The project is moving not because of the technical achievement

but because the choice of hardware *means* something. The obsolescence is the point. The fragility is the beauty.

The practical implication: technique is necessary but not sufficient. You can master every effect in this book and still produce a demo nobody remembers. What makes a demo memorable is not what it does but what it says. Even an abstract demo has a personality: its pacing says something about tension and release; its colour palette evokes a mood; its music choice creates emotional context. The coder who treats these as afterthoughts produces a tech demo. The coder who treats them as design decisions produces a demo.

Lo-Fi Motion embraced its lo-fi aesthetic as identity. Eager turned the 32x24 grid from constraint into creative choice. NHBF found beauty in the puzzle of 256 bytes. In each case, the limitation became the medium.

This is what "MORE" demands. Not more polygons, not more colours, not more effects. More ambition. More intention. More willingness to treat a ZX Spectrum demo as an art form.

20.9 Your First Demo: A Practical Roadmap

For the reader who has followed this book from Chapter 1 and wants to make a demo, here is a concrete path.

"'mermaid id:ch20_your_first_demo_a_practical graph TD IDEA["Idea(visual concept, mood, music)"] –> PROTO["Prototype(verify/ HTML/JS, orquick Z80 test)"] PROTO –> IMPL["Z80 Implementation(sjasmplus, effect code)"] IMPL –> TIME{"Fits inframe budget?"} TIME – No –> OPT["Optimise(unroll, precompute,reduce scope)"] OPT –> IMPL TIME – Yes –> POLISH["Polish(transitions, sync to music,colour palette)"] POLISH –> PARTY["Party Version(submit to compo)"] PARTY –> FINAL["Final Version(fix bugs, add credits,test on hardware)"]

```
IMPL -.-> |"border-colour timing"| TIME
POLISH -.-> |"scene table reorder"| POLISH

style IDEA fill:#ffd,stroke:#993
style PROTO fill:#ddf,stroke:#339
style IMPL fill:#dfd,stroke:#393
style OPT fill:#fdd,stroke:#933
style PARTY fill:#fdf,stroke:#939
style FINAL fill:#dff,stroke:#399


> **The iterative loop:** The path from implementation to timing check and back is where most

### Week 1: Foundation

1. **Set up the toolchain.** Install sjasmplus, choose an emulator (Unreal Speccy, Fuse, or ZE

2. **Build the scene table engine.** Write a minimal engine that reads a scene table and calls
Fi Motion's architecture: bank number, entry address, frame count. Get it working with a singl
```

3. **Add music.** Integrate a PT3 player into your IM2 interrupt handler (Chapter 11). Drop in

Week 2: Effects

4. **Build your first real effect.** Attribute plasma is the natural starting point -- it is c
understood (Chapter 9). Render into a virtual buffer and copy to attribute RAM.

5. **Build your second effect.** Fire, rotozoomer, rain, colour bars -- pick one from the effe

6. **Add a transition.** A simple crossfade between two attribute buffers: interpolate the col
50 frames. Or a hard cut synchronised to a beat in the music.

Week 3: Polish

7. **Replace placeholder music.** If you have a musician collaborator, integrate the final tra

8. **Adjust timing.** This is where the scene table earns its value. Rearrange effects, adjust

9. **Add a beginning and an end.** A loading screen (compressed with ZX0, Chapter 14), an open

Week 4: Release

10. **Test.** Multiple emulators. Cold boot. The exact file you will submit.

11. **Write the NFO.** Credits, platform requirements, greetings, and -- if you are feeling ge

12. **Submit.** Choose a party. Follow the rules. Upload the file. Then watch the compo and en

Your first entry is unlikely to place. Treat it as a learning exercise: the feedback from seei

---

Summary

- **Design is everything.** Introspec defines demo design as "the complete aggregate of all de

- **Lo-Fi Motion provides a replicable production template:** a scene table drives the demo st
byte-per-pixel buffers, and the toolchain (sjasmplus + zemu + Ruby scripts + hrust1opt) chains

- **The scene table pattern** separates content from engine. Adding, removing, or reordering e

- **Making-of culture is a strength of the ZX scene.** Detailed technical writeups -- from Eag
editor workflow to NHBF's 256-byte puzzle -- serve as education, documentation, and community

- **The standard toolchain** converges on sjasmplus (assembler), Unreal Speccy or Fuse (emulat

- **Compo culture** centres on events like Chaos Constructions, DiHalt, Multimatograf, CAFe, a

- **The community** lives on Hype (technical discussion, making-of articles), ZXArt (productio

- **Project management matters.** Build the minimum viable demo first, then iterate. Establish

```
- **Introspec's "MORE"** challenges demo makers to transcend platform limitations: "Two pixels

---

*Next: Chapter 21 -- Full Game: ZX Spectrum 128K. We move from demos to games, integrating eve
scrolling platformer.*

> **Sources:** restorer, "Making of Lo-Fi Motion," Hype, 2020 (hype.retroscene.org/blog/demo/1
of," Hype, 2025 (hype.retroscene.org/blog/dev/1120.html)


\newpage

# Chapter 21: Full Game -- ZX Spectrum 128K

> *"The only way to know if your engine works is to ship a game."*

---

You have sprites (Chapter 16). You have scrolling (Chapter 17). You have a game loop and entit

Now you must put all of it into a single binary that loads from tape, shows a loading screen,
scrolling platformer with four enemy types and a boss, tracks high scores, and fits on a `.tap

This is the integration chapter. No new techniques appear here. Instead, we face the problems

The game we are building is called *Ironclaw* -- a five-level side-scrolling platformer starri
scrolling platformers demand every subsystem simultaneously and leave nowhere to hide. If the

---

## 21.1 Project Architecture

Before writing a single line of Z80, you need a directory structure that scales. A 128K game w

### Directory Layout

```text
ironclaw/
  src/
    main.a80           -- entry point, bank switching, state machine
    render.a80          -- tile renderer, scroll engine
    sprites.a80         -- sprite drawing routines (OR+AND masked)
    entities.a80        -- entity update, spawning, despawning
    physics.a80         -- gravity, friction, jump, collision response
    collisions.a80      -- AABB and tile collision checks
    ai.a80              -- enemy FSM: patrol, chase, attack, retreat, death
    player.a80          -- player input, state, animation
    hud.a80             -- score, lives, status bar
    menu.a80            -- title screen, options, high scores
```
```

```
    loader.a80          -- loading screen, tape/esxDOS loader
    music_driver.a80    -- PT3 player, interrupt handler
    sfx.a80             -- sound effects engine, channel stealing
    esxdos.a80          -- DivMMC file I/O wrappers
    banks.a80           -- bank switching macros and utilities
    defs.a80            -- constants, memory map, entity structure
  data/
    levels/             -- level tilemaps (compressed)
    tiles/              -- tileset graphics
    sprites/            -- sprite sheets (pre-shifted)
    music/              -- PT3 music files
    sfx/                -- SFX definition tables
    screens/            -- loading screen, title screen
  tools/
    png2tiles.py        -- PNG tileset converter
    png2sprites.py      -- PNG sprite sheet converter (generates shifts)
    map2bin.py          -- Tiled JSON/TMX to binary tilemap
    compress.py         -- wrapper around ZX0/Pletter compression
  build/                -- compiled output (gitignored)
  Makefile              -- the build system
```

Every source file focuses on one subsystem. Every data file passes through a conversion pipeline before it reaches the assembler. The `tools/` directory holds Python scripts that convert artist-friendly formats (PNG images, Tiled editor maps) into assembler-ready binary data.

The Build System

The Makefile is the spine of the project. It must:

1. Convert all graphics from PNG to binary tile/sprite data
2. Convert level maps from Tiled format to binary tilemaps
3. Compress level data, graphics banks, and music with the appropriate compressor
4. Assemble all source files into a single binary
5. Generate the final `.tap` file with the correct loader

```
# Ironclaw Makefile
ASM       = sjasmplus
COMPRESS  = zx0
PYTHON    = python3

# Data conversion
data/tiles/tileset.bin: data/tiles/tileset.png
    $(PYTHON) tools/png2tiles.py $< $@

data/sprites/player.bin: data/sprites/player.png
    $(PYTHON) tools/png2sprites.py --shifts 4 $< $@

data/levels/level%.bin: data/levels/level%.tmx
    $(PYTHON) tools/map2bin.py $< $@

# Compression (ZX0 for level data -- good ratio, small decompressor)
```

```
data/levels/level%.bin.zx0: data/levels/level%.bin
    $(COMPRESS) $< $@

# Compression (Pletter for graphics -- faster decompression)
data/tiles/tileset.bin.plt: data/tiles/tileset.bin
    pletter5 $< $@

# Assembly
build/ironclaw.tap: src/*.a80 data/levels/*.zx0 data/tiles/*.plt \
                    data/sprites/*.bin data/music/*.pt3
    $(ASM) --fullpath src/main.a80 --raw=build/ironclaw.tap

.PHONY: clean
clean:
    rm -rf build/ data/**/*.bin data/**/*.zx0 data/**/*.plt
```

The key insight is the data pipeline. The artist exports a PNG tileset from Aseprite. The `png2tiles.py` script slices it into 8x8 or 16x16 tiles, converts each to the Spectrum's interleaved pixel format, and writes a binary blob. The level designer exports a Tiled `.tmx` map. The `map2bin.py` script extracts the tile indices and writes a compact binary tilemap. The compressor squeezes each blob. Only then does the assembler `INCBIN` the result into the appropriate memory bank.

This pipeline means the game's content is always in editable form (PNG, TMX), and the build system handles every conversion automatically. Change a tile in the PNG, type `make`, and the new tile appears in the game.

21.2 Memory Map: 128K Bank Assignments

The ZX Spectrum 128K has eight 16KB RAM banks, numbered 0 through 7. At any moment, the CPU sees a 64KB address space:

```
$0000-$3FFF   ROM (16KB) -- BASIC or 128K editor ROM
$4000-$7FFF   Bank 5 (always) -- screen memory (normal screen)
$8000-$BFFF   Bank 2 (always) -- typically code
$C000-$FFFF   Switchable -- banks 0-7, selected via port $7FFD
```

Banks 5 and 2 are hardwired to `$4000` and `$8000` respectively. Only the top 16KB window (`$C000-$FFFF`) is switchable. The bank selection register at port `$7FFD` also controls which screen is displayed (bank 5 or bank 7) and which ROM page is active.

"'z80 id:ch21_memory_map_128k_bank_2 ; Port $7FFD layout: ; Bit 0-2: Bank number for $C000—FFFF (0-7) ; Bit 3: Screen select (0 = bank 5 normal, 1 = bank 7 shadow) ; Bit 4: ROM select (0 = 128K editor, 1 = 48K BASIC) ; Bit 5: Disable paging (PERMANENT – cannot be undone without reset) ; Bits 6-7: Unused

; Switch to bank N at $C000 ; Input: A = bank number (0-7) ; Preserves: all registers except A switch_bank: or %00010000 ; ROM 1 (48K BASIC) – keep this set ld (last_bank_state), a ld bc, $7FFD out (c), a ret

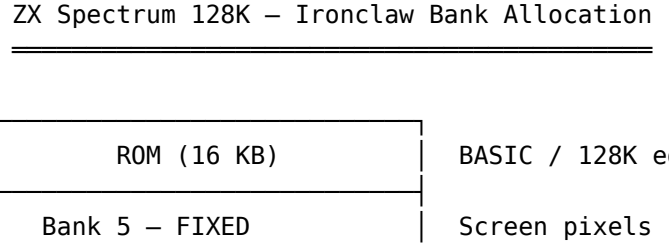
last_bank_state: db %00010000 ; default: bank 0, normal screen, ROM 1

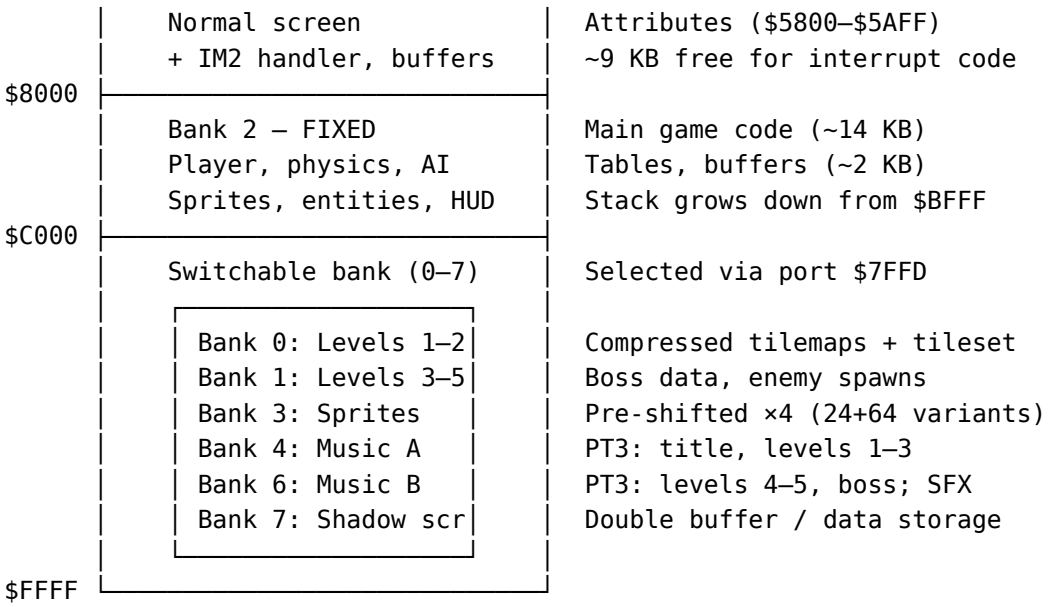
The critical rule: **always store your last write to `$7FFD`** in a shadow variable. Port `$7F
only -- you cannot read back the current state. If you need to change one bit (say, switch the

Ironclaw Bank Allocation

Here is how Ironclaw distributes its 128KB across the eight banks:

```text
Bank 0 ($C000)  -- Level data: tilemaps for levels 1-2 (compressed)
                   Tileset graphics (compressed)
                   Decompression buffer

Bank 1 ($C000)  -- Level data: tilemaps for levels 3-5 (compressed)
                   Boss level data and patterns
                   Enemy spawn tables

Bank 2 ($8000)  -- FIXED: Main game code
                   Player logic, physics, collisions
                   Sprite routines, entity system
                   State machine, HUD
                   ~ 14KB code, 2KB tables/buffers

Bank 3 ($C000)  -- Sprite graphics (pre-shifted x4)
                   Player: 6 frames x 4 shifts = 24 variants
                   Enemies: 4 types x 4 frames x 4 shifts = 64 variants
                   Projectiles, particles, pickups
                   ~ 12KB total

Bank 4 ($C000)  -- Music: PT3 song data (title, levels 1-3)
                   PT3 player code (resident copy)

Bank 5 ($4000)  -- FIXED: Normal screen
                   Pixel data $4000-$57FF (6,144 bytes)
                   Attributes $5800-$5AFF (768 bytes)
                   Remaining ~9KB: interrupt handler, screen buffers

Bank 6 ($C000)  -- Music: PT3 song data (levels 4-5, boss, game over)
                   SFX definition tables
                   SFX engine code

Bank 7 ($4000)  -- Shadow screen (used for double buffering)
                   Also usable as 16KB data storage when
                   not actively double-buffering
```





Several things to notice about this layout:

Code lives in bank 2 (fixed). Because bank 2 is always mapped at `$8000-$BFFF`, your main game code is always accessible. You never need to page in code – only data. This eliminates the most dangerous class of banking bug: calling a routine that has been paged out.

Sprite graphics in bank 3, separate from level data in banks 0-1. When rendering a frame, the renderer needs both tile graphics (for the scrolling background) and sprite graphics (for entities). If both were in the same switchable bank, you would need to page back and forth mid-render. By placing them in separate banks, you can page in the tile data, render the background, then page in the sprite data and render all entities, with only two bank switches per frame.

Music is split across banks 4 and 6. The PT3 player runs during the interrupt handler, which fires once per frame. The interrupt handler must page in the music bank, update the AY registers, and page back to whatever bank the main loop was using. Splitting music across two banks means the interrupt handler must know which bank contains the current song. We handle this with a variable:

"'z80 id:ch21_ironclaw_bank_allocation_3 current_music_bank: db 4 ; bank 4 by default

im2_handler: push af push bc push de push hl push ix push iy ; IY must be preserved – BASIC uses it ; for system variables, and PT3 players ; typically use IY internally

```
; Save current bank state
ld   a, (last_bank_state)
push af

; Page in music bank
ld   a, (current_music_bank)
call switch_bank
```

```
; Update PT3 player -- writes AY registers
call pt3_play

; Check for pending SFX
call sfx_update

; Restore previous bank
pop  af
ld   (last_bank_state), a
ld   bc, $7FFD
out  (c), a

pop  iy
pop  ix
pop  hl
pop  de
pop  bc
pop  af
ei
reti
```

The shadow screen in bank 7 is available for double-buffering during scroll updates (as de
buffering -- during the menu, between levels, during cutscenes -- bank 7 is 16KB of free stora

The Stack

The stack lives at the top of bank 2's address space, growing downward from `$BFFF`. With ~14K
based sprite output (Chapter 16's PUSH method), remember that you are borrowing the stack poin

---

21.3 The State Machine

A game is not one program. It is a sequence of modes -- title screen, menu, gameplay, pause, g

```z80 id:ch21_the_state_machine
; Game states
STATE_LOADER    equ  0
STATE_TITLE     equ  1
STATE_MENU      equ  2
STATE_GAMEPLAY  equ  3
STATE_PAUSE     equ  4
STATE_GAMEOVER  equ  5
STATE_HISCORE   equ  6
STATE_LEVELWIN  equ  7

; State handler table -- each entry is a 2-byte address
state_table:
    dw   state_loader       ; 0: loading screen + init
    dw   state_title        ; 1: title screen with animation
```

```
    dw   state_menu         ; 2: main menu (start, options, hiscores)
    dw   state_gameplay     ; 3: in-game
    dw   state_pause        ; 4: paused
    dw   state_gameover     ; 5: game over sequence
    dw   state_hiscore      ; 6: high score entry
    dw   state_levelwin     ; 7: level complete, advance

current_state:
    db   STATE_LOADER

; Main loop -- called once per frame after HALT
main_loop:
    halt                    ; wait for frame interrupt

    ; Dispatch to current state handler
    ld   a, (current_state)
    add  a, a               ; x2 for word index
    ld   l, a
    ld   h, 0
    ld   de, state_table
    add  hl, de
    ld   a, (hl)
    inc  hl
    ld   h, (hl)
    ld   l, a               ; HL = handler address
    jp   (hl)               ; jump to handler

; Each handler ends with:  jp main_loop
```

Each state handler owns the frame completely. The gameplay handler runs input, physics, AI, rendering, and HUD updates. The menu handler reads input and draws the menu. The pause handler simply waits for the unpause key, displaying a "PAUSED" overlay.

State transitions happen by writing a new value to `current_state`. The transition from `STATE_GAMEPLAY` to `STATE_PAUSE` requires no cleanup – the game state is untouched, and returning to `STATE_GAMEPLAY` resumes exactly where you left off. But the transition from `STATE_GAMEOVER` to `STATE_HISCORE` requires checking whether the player's score qualifies, and the transition from `STATE_LEVELWIN` to `STATE_GAMEPLAY` requires loading and decompressing the next level's data.

21.4 The Gameplay Frame

This is where integration happens. During `STATE_GAMEPLAY`, every frame must execute the following, in order:

```
1. Read input                ~200 T-states
2. Update player physics     ~800 T-states
3. Update player state       ~400 T-states
4. Update enemies (AI+phys)  ~4,000 T-states (8 enemies)
5. Check collisions          ~2,000 T-states
```

```
6. Update projectiles        ~500 T-states
7. Scroll the viewport       ~8,000-15,000 T-states (depends on method)
8. Render background tiles   ~12,000 T-states (exposed column/row)
9. Erase old sprites         ~3,000 T-states (background restore)
10. Draw sprites             ~8,000 T-states (8 entities x ~1,000 each)
11. Update HUD               ~1,500 T-states
12. [Music plays in IM2]     ~3,000 T-states (interrupt handler)
                             ──────────────

                   Total:    ~43,400-50,400 T-states
```

On a Pentagon with 71,680 T-states per frame, that leaves 21,000-28,000 T-states of slack. That sounds comfortable, but it is deceptive. Those estimates are averages. When four enemies are on screen and the player is jumping across a gap with projectiles flying, the worst case can spike 20-30% above the average. Your slack is your safety margin.

The order matters. Input must come first – you need the player's intent before simulating physics. Physics must precede collision detection – you need to know where entities want to move before checking whether they can. Collision response must precede rendering – you need final positions before drawing anything. And sprites must be drawn after the background, because sprites overlay the tiles.

Reading Input

"'z80 id:ch21_reading_input ; Read keyboard and Kempston joystick ; Returns result in A: bit 0=right, 1=left, 2=down, 3=up, 4=fire read_input: ld d, 0 ; accumulate result

```
; Kempston joystick (active high)
in   a, ($1F)          ; Kempston port
and  %00011111         ; mask 5 bits: fire, up, down, left, right
ld   d, a

; Keyboard: QAOP + space (merge with joystick)
; Q = up
ld   bc, $FBFE         ; half-row Q-T
in   a, (c)
bit  0, a              ; Q key
jr   nz, .not_q
set  3, d              ; up
```

.not_q: ; O = left ld b, $DF ; half-row Y-P in a, (c) bit 1, a ; O key jr nz, .not_o set 1, d ; left .not_o: ; P = right bit 0, a ; P key (same half-row) jr nz, .not_p set 0, d ; right .not_p: ; A = down ld b, $FD ; half-row A-G in a, (c) bit 0, a ; A key jr nz, .not_a set 2, d ; down .not_a: ; Space = fire ld b, $7F ; half-row space-B in a, (c) bit 0, a ; space jr nz, .not_fire set 4, d ; fire .not_fire:

```
ld   a, d
ld   (input_state), a
ret
```

```
Note that keyboard reads use `IN A,(C)` with the half-row address in B. Each key maps to a bit
```

```
### The Scroll Engine

Scrolling is the most expensive operation in the frame. Chapter 17 covered the techniques in d

Ironclaw uses the **combined scroll** method: character-granularity scrolling (8-
pixel jumps) for the main viewport, with a pixel offset (0-7) within the current 8-
pixel window for smooth visual movement. When the pixel offset reaches 8, the viewport shifts

The viewport is 30 characters wide (240 pixels) and 20 characters tall (160 pixels), leaving r
character HUD at the top and bottom. The level tilemap is typically 256-512 tiles wide and 20

When the viewport shifts by one tile column, the renderer must:

1. Copy 29 columns of the current screen one character left (or right)
2. Draw the newly exposed column of tiles from the tilemap

The column copy is an LDIR chain: 20 rows x 8 pixel lines x 29 bytes = 4,640 bytes at 21 T-
states each = 97,440 T-states. That is more than an entire frame. This is why Chapter 17's sha

```z80 id:ch21_the_scroll_engine
; Shadow screen double-buffer scroll
; Frame N: display screen is bank 5, draw screen is bank 7
; 1. Draw the shifted background into bank 7
; 2. Flip: set bit 3 of $7FFD to display bank 7
; Frame N+1: display screen is bank 7, draw screen is bank 5
; 3. Draw the shifted background into bank 5
; 4. Flip: clear bit 3 of $7FFD to display bank 5

flip_screen:
    ld   a, (last_bank_state)
    xor  %00001000          ; toggle screen bit (bit 3)
    ld   (last_bank_state), a
    ld   bc, $7FFD
    out  (c), a
    ret
```

But even with double-buffering, the full column copy is expensive. Ironclaw optimises this by spreading the work: during smooth sub-tile scrolling (pixel offset 1-7), there is no column copy – only the offset changes. The expensive column copy happens only on tile boundaries, roughly every 4-8 frames depending on the player's speed. Between those spikes, scroll rendering is nearly free.

When a tile boundary is crossed, the column copy can be spread across two frames using the double buffer: Frame N draws the top half of the shifted screen into the back buffer, Frame N+1 draws the bottom half and flips. The player sees a seamless scroll because the flip only happens when the back buffer is complete.
```

21.5 Sprite Integration

Ironclaw uses OR+AND masked sprites (Chapter 16, method 2) for all game entities. This is the standard technique: for each sprite pixel, AND with a mask byte to clear the background, then OR with the sprite data to set the pixels.

Each 16x16 sprite has four pre-shifted copies (Chapter 16, method 3), one for each 2-pixel horizontal alignment. This reduces per-pixel shifting from a runtime operation to a table lookup. The cost: each sprite frame requires 4 variants x 16 lines x 3 bytes/line (2 bytes data + 1 byte mask, widened to 3 bytes to handle shift overflow) = 192 bytes. But the rendering speed drops from ~1,500 T-states to ~1,000 T-states per sprite, and with 8-10 sprites on screen, that savings adds up.

Pre-shifted sprite data lives in bank 3. During the sprite rendering phase, the renderer pages in bank 3, iterates through all active entities, and draws each one:

“‘z80 id:ch21_sprite_integration ; Draw all active entities ; Assumes bank 3 (sprite graphics) is paged in at $C000 render_entities: ld ix, entity_array ld b, MAX_ENTITIES

.loop: push bc

```
; Check if entity is active
ld   a, (ix + ENT_FLAGS)
bit  FLAG_ACTIVE, a
jr   z, .skip

; Calculate screen position from world position and viewport
ld   l, (ix + ENT_X)
ld   h, (ix + ENT_X + 1)
ld   de, (viewport_x)
or   a                  ; clear carry
sbc  hl, de             ; screen_x = world_x - viewport_x
; Check if on screen (0-239)
bit  7, h
jr   nz, .skip          ; off-screen left (negative)
ld   a, h
or   a
jr   nz, .skip          ; off-screen right (> 255)
ld   a, l
cp   240
jr   nc, .skip          ; off-screen right (240-255)

; Store screen X for sprite routine
ld   (sprite_screen_x), a

; Y position (already in screen coordinates for simplicity)
ld   a, (ix + ENT_Y)
ld   (sprite_screen_y), a

; Look up sprite graphic address from type + frame + shift
call get_sprite_address ; returns HL = address in bank 3

; Draw masked sprite at (sprite_screen_x, sprite_screen_y)
```

```
call draw_sprite_masked
```

.skip: pop bc ld de, ENT_SIZE add ix, de ; next entity djnz .loop ret

```
### Background Restore (Dirty Rectangles)

Before drawing sprites at their new positions, you must erase them from their old positions. I

```z80 id:ch21_background_restore_dirty
; Dirty rectangle entry: 4 bytes
;   byte 0: screen address low
;   byte 1: screen address high
;   byte 2: width in bytes
;   byte 3: height in pixel lines

; Save background before drawing sprite
save_background:
    ; HL = screen address, B = height, C = width
    ld   de, bg_save_buffer
    ld   (bg_save_ptr), de
    ; ... copy rectangle from screen to buffer ...
    ret

; Restore all saved backgrounds (called before new sprite render pass)
restore_backgrounds:
    ld   hl, dirty_rect_list
    ld   b, (hl)           ; count of dirty rectangles
    inc  hl
    or   a
    ret  z                 ; no sprites last frame

.loop:
    push bc
    ; Read rectangle descriptor
    ld   e, (hl)
    inc  hl
    ld   d, (hl)           ; DE = screen address
    inc  hl
    ld   b, (hl)           ; B = height
    inc  hl
    ld   c, (hl)           ; C = width
    inc  hl
    push hl

    ; Copy saved background back to screen
    ; ... copy from bg_save_buffer to screen ...

    pop  hl
    pop  bc
    djnz .loop
    ret
```
```

The cost of dirty rectangles is proportional to the number and size of sprites. For 8 entities of 16x16 pixels (3 bytes wide after shift), saving and restoring costs roughly 8 x 16 x 3 x 2 (save + restore) x ~10 T-states/byte = ~7,680 T-states. Not cheap, but predictable.

21.6 Collisions, Physics, and AI in Context

Chapters 18 and 19 covered these systems in isolation. In the integrated game, the key challenge is ordering: which system runs first, and what data does each need from the others?

Physics-Collision Loop

The physics update must interleave with collision detection. The pattern is:

```
1. Apply gravity:  velocity_y += GRAVITY
2. Apply input:    if (input_right) velocity_x += ACCEL
3. Horizontal move:
     a. new_x = x + velocity_x
     b. Check tile collisions at (new_x, y)
     c. If blocked: push back to tile boundary, velocity_x = 0
     d. Else: x = new_x
4. Vertical move:
     a. new_y = y + velocity_y
     b. Check tile collisions at (x, new_y)
     c. If blocked: push back, velocity_y = 0, set on_ground flag
     d. Else: y = new_y, clear on_ground flag
5. If (on_ground AND input_jump): velocity_y = -JUMP_FORCE
```

The horizontal and vertical moves are separate because collision response must handle each axis independently. If you move diagonally and hit a corner, you want to slide along the wall on one axis while stopping on the other. Checking both axes simultaneously leads to "sticking" bugs where the player gets trapped on corners.

All positions use 8.8 fixed-point format (Chapter 4): the high byte is the pixel coordinate, the low byte is the fractional part. Velocity values are also 8.8. This gives sub-pixel movement precision without requiring any multiplication in the core physics loop – addition and shifting are sufficient.

```
z80 id:ch21_physics_collision_loop_2 ; Apply gravity to entity at IX ; velocity_y
is 16-bit signed, 8.8 fixed-point apply_gravity:     ld   l, (ix + ENT_VY)
ld   h, (ix + ENT_VY + 1)     ld   de, GRAVITY       ; e.g., $0040 = 0.25 pixels/frame/frame
add  hl, de     ; Clamp to terminal velocity     ld   a, h     cp   MAX_FALL_SPEED
; e.g., 4 pixels/frame     jr   c, .no_clamp     ld   hl, MAX_FALL_SPEED * 256
.no_clamp:     ld   (ix + ENT_VY), l     ld   (ix + ENT_VY + 1), h     ret
```

Tile Collision

The tile collision check converts a pixel coordinate to a tile index, then looks up the tile type in the level's collision map:

“‘z80 id:ch21_tile_collision ; Check tile at pixel position (B=x, C=y) ; Returns: A = tile type (0=empty, 1=solid, 2=hazard, 3=platform) check_tile: ; Convert pixel X to tile column: x / 8 ld a, b srl a srl a srl a ; A = column (0-31) ld l, a

```
; Convert pixel Y to tile row: y / 8
ld   a, c
srl  a
srl  a
srl  a                ; A = row (0-23)

; Tile index = row * level_width + column
ld   h, 0
ld   d, h
ld   e, a
; Multiply row by level_width (e.g., 256 = trivial: just use E as high byte)
; For level_width = 256: address = level_map + row * 256 + column
ld   d, e             ; D = row = high byte of offset
ld   e, l             ; E = column = low byte of offset
ld   hl, level_collision_map
add  hl, de
ld   a, (hl)          ; A = tile type
ret


For Ironclaw, level widths are set to 256 tiles. This is not a coincidence -- it makes the row
tile-wide level at 8 pixels per tile is 2,048 pixels, roughly 8.5 screens wide. For longer lev
tile width (multiply row by 2 via `SLA E : RL D`), though this costs a few extra T-
states per lookup.

### Enemy AI

Each enemy type has a finite state machine (Chapter 19). The state is stored in the entity str

```z80 id:ch21_enemy_ai
; Entity structure (16 bytes per entity)
ENT_X       equ  0    ; 16-bit, 8.8 fixed-point
ENT_Y       equ  2    ; 16-bit, 8.8 fixed-point
ENT_VX      equ  4    ; 16-bit, 8.8 fixed-point
ENT_VY      equ  6    ; 16-bit, 8.8 fixed-point
ENT_TYPE    equ  8    ; entity type (player, walker, flyer, shooter, boss)
ENT_STATE   equ  9    ; FSM state (idle, patrol, chase, attack, retreat, dying)
ENT_ANIM    equ  10   ; animation frame counter
ENT_HEALTH  equ  11   ; hit points
ENT_FLAGS   equ  12   ; bit flags: active, on_ground, facing_left, invuln, ...
ENT_TIMER   equ  13   ; general-purpose timer (attack cooldown, etc.)
ENT_AUX1    equ  14   ; type-specific data (patrol point, projectile type, etc.)
ENT_AUX2    equ  15   ; type-specific data
ENT_SIZE    equ  16

MAX_ENTITIES equ 16   ; player + 8 enemies + 7 projectiles
```

Ironclaw’s four enemy types:
```

1. **Walker** – Patrols between two points. When the player is within 64 pixels horizontally, switches to Chase state (walks toward player). Switches to Attack (contact damage) on collision. Returns to Patrol when player moves away or the enemy reaches a ledge.
2. **Flyer** – Sine-wave vertical movement (using the sine table from Chapter 4). Ignores tile collisions. Chases the player horizontally when within range. Drops projectiles at intervals.
3. **Shooter** – Stationary. Fires a horizontal projectile every N frames when the player is within line-of-sight (same tile row, no solid tiles between). The projectile is a separate entity allocated from the entity pool.
4. **Boss** – Multi-phase FSM. Phase 1: patrol platform, fire spread shots. Phase 2 (below 50% health): faster movement, targeted shots, summons walkers. Phase 3 (below 25% health): enrage, continuous fire, screen shake.

The key optimisation from Chapter 19: AI does not run every frame. Enemy AI updates are distributed across frames using a simple round-robin:

“‘z80 id:ch21_enemy_ai_2 ; Update AI for subset of enemies each frame ; ai_frame_counter cycles 0, 1, 2, 0, 1, 2, ... update_enemy_ai: ld a, (ai_frame_counter) inc a cp 3 jr c, .no_wrap xor a .no_wrap: ld (ai_frame_counter), a

```
; Only update enemies where (entity_index % 3) == ai_frame_counter
ld   ix, entity_array + ENT_SIZE  ; skip player (index 0)
ld   b, MAX_ENTITIES - 1
ld   c, 0              ; entity index counter
```

.loop: push bc ld a, (ix + ENT_FLAGS) bit FLAG_ACTIVE, a jr z, .next

```
; Check if this entity's turn
ld   a, c
ld   e, 3
call mod_a_e           ; A = entity_index % 3
ld   b, a
ld   a, (ai_frame_counter)
cp   b
jr   nz, .next

; Run AI for this entity
call run_entity_ai     ; dispatch based on ENT_TYPE and ENT_STATE
```

.next: pop bc inc c ld de, ENT_SIZE add ix, de djnz .loop ret

```
This means each enemy's AI runs once every 3 frames. At 50 fps, that is still ~17 AI updates p
states per enemy, running all 8 enemies every frame costs 4,000 T-states. Running 2-
3 enemies per frame costs 1,000-1,500 T-states. Physics and collision detection still run ever

---

## 21.7 Sound Integration

### Music
```

The PT3 player runs inside the IM2 interrupt handler, as shown in section 21.2. The player occ
2KB of code and executes once per frame, taking ~2,500-3,500 T-states depending on the complex

Each level has its own music track. When transitioning between levels, the game:

1. Fades out the current track (ramp AY volumes to 0 over 25 frames)
2. Pages in the appropriate music bank (bank 4 or 6)
3. Initialises the PT3 player with the new song's start address
4. Fades in

The PT3 data format is compact -- a typical 2-3 minute game music loop compresses to 2-
4KB with Pletter, which is why two music banks (4 and 6) can hold all six tracks (title, five

Sound Effects

Sound effects use the priority-based channel stealing system from Chapter 11. When a sound eff

```z80 id:ch21_sound_effects
; SFX priority levels
SFX_JUMP       equ  1     ; low priority
SFX_PICKUP     equ  2
SFX_SHOOT      equ  3
SFX_HIT        equ  4
SFX_EXPLODE    equ  5     ; high priority
SFX_BOSS_DIE   equ  6     ; highest priority

; Trigger a sound effect
; A = SFX id
play_sfx:
    ; Check priority -- only play if higher than current SFX
    ld   hl, current_sfx_priority
    cp   (hl)
    ret  c                ; current SFX has higher priority, ignore

    ; Set up SFX playback
    ld   (hl), a          ; update priority
    ; Look up SFX descriptor table
    add  a, a             ; x2 for word index
    ld   l, a
    ld   h, 0
    ld   de, sfx_table
    add  hl, de
    ld   a, (hl)
    inc  hl
    ld   h, (hl)
    ld   l, a             ; HL = SFX descriptor address

    ; SFX descriptor: duration (byte), channel (byte),
    ;                 then per-frame: freq_lo, freq_hi, volume, noise
    ld   a, (hl)
    ld   (sfx_frames_left), a
```

```
    inc  hl
    ld   a, (hl)
    ld   (sfx_channel), a
    inc  hl
    ld   (sfx_data_ptr), hl
    ret
```

The SFX update runs inside the interrupt handler, after the PT3 player. If an SFX is active, it overwrites the AY register values that the PT3 player just set for the stolen channel. This means the music continues to play correctly on the other two channels, and the stolen channel produces the sound effect.

SFX definitions are procedural tables rather than sampled audio. Each entry is a sequence of per-frame register values:

```
z80 id:ch21_sound_effects_2 ; SFX: player jump -- ascending frequency sweep
on channel C sfx_jump_data:     db   8                 ; duration: 8 frames
db   2                  ; channel C (0=A, 1=B, 2=C)     ; Per-frame: freq_lo,
freq_hi, volume     db   $80, $01, 15      ; frame 1: low pitch, full volume
db   $60, $01, 14      ; frame 2: slightly higher     db   $40, $01, 13     db
$20, $01, 12     db   $00, $01, 10     db   $E0, $00, 8     db   $C0, $00, 5
db   $A0, $00, 2       ; frame 8: high pitch, fading out
```

This approach uses negligible memory (8-20 bytes per effect) and negligible CPU time (a few dozen T-states per frame to write 3-4 AY register values).

21.8 Loading: Tape and DivMMC

A ZX Spectrum game must load somehow. In the 1980s, that meant tape. Today, most users have a DivMMC (or similar) SD card interface running esxDOS. Ironclaw supports both.

The .tap File and BASIC Loader

The `.tap` file format is a sequence of data blocks, each preceded by a 2-byte length and a flag byte. A BASIC loader program (itself a block in the .tap) uses `LOAD "" CODE` commands to load each block to the correct address.

Ironclaw's .tap structure:

```
Block 0:  BASIC loader program (autorun line 10)
Block 1:  Loading screen (6912 bytes -> $4000)
Block 2:  Main code block (bank 2 content -> $8000)
Block 3:  Bank 0 data (level data + tiles, compressed)
Block 4:  Bank 1 data (more level data)
Block 5:  Bank 3 data (sprite graphics)
Block 6:  Bank 4 data (music tracks 1-3)
Block 7:  Bank 6 data (music tracks 4-6, SFX)
```

The BASIC loader:

```
10 CLEAR 32767
20 LOAD "" SCREEN$
```

```
30 LOAD "" CODE
40 BORDER 0: PAPER 0: INK 0: CLS
50 RANDOMIZE USR 32768
```

Line 10 sets RAMTOP below `$8000`, protecting our code from BASIC's stack. Line 20 loads the loading screen directly into screen memory (the Spectrum's `LOAD "" SCREEN$` does this automatically). Line 30 loads the main code block. Line 40 clears the screen. Line 50 jumps to our code at `$8000`.

But this only loads the main code block. The banked data (blocks 3-7) must be loaded by our own Z80 code, which pages in each bank and uses the ROM's tape loading routine:

"'z80 id:ch21_the_tap_file_and_basic_loader_3 ; Load bank data from tape ; Called after main code is running load_bank_data: ; Bank 0 ld a, 0 call switch_bank ld ix, $C000 ; load address ld de, BANK0_SIZE ; data length call load_tape_block

```
; Bank 1
ld   a, 1
call switch_bank
ld   ix, $C000
ld   de, BANK1_SIZE
call load_tape_block

; ... repeat for banks 3, 4, 6 ...
ret
```

; Load one tape block using ROM routine ; IX = address, DE = length load_tape_block: ld a, $FF ; data block flag (not header) scf ; carry set = LOAD (not VERIFY) call $0556 ; ROM tape loading routine ret nc ; carry clear = load error ret

esxDOS Loading (DivMMC)

For users with a DivMMC or similar hardware, loading from an SD card is dramatically faster an

```z80 id:ch21_esxdos_loading_divmmc
; esxDOS function codes
F_OPEN      equ  $9A
F_CLOSE     equ  $9B
F_READ      equ  $9D
F_WRITE     equ  $9E
F_SEEK      equ  $9F
F_OPENDIR   equ  $A3
F_READDIR   equ  $A4

; esxDOS open modes
FA_READ     equ  $01
FA_WRITE    equ  $06
FA_CREATE   equ  $0E

; Open a file
; IX = pointer to null-terminated filename
```

```
; Returns: A = file handle (or carry set on error)
esx_open:
    ld   a, '*'            ; use default drive
    ld   b, FA_READ        ; open for reading
    rst  $08
    db   F_OPEN
    ret

; Read bytes from file
; A = file handle, IX = destination address, BC = byte count
; Returns: BC = bytes actually read (or carry set on error)
esx_read:
    rst  $08
    db   F_READ
    ret

; Close a file
; A = file handle
esx_close:
    rst  $08
    db   F_CLOSE
    ret
```

Ironclaw detects whether esxDOS is present at startup by checking for the DivMMC signature. If present, it loads all data from files on the SD card instead of tape:

“‘z80 id:ch21_esxdos_loading_divmmc_2 ; Load game data from esxDOS ; All bank data stored in separate files on SD card load_from_esxdos: ; Load bank 0: levels + tiles ld a, 0 call switch_bank ld ix, filename_bank0 call esx_open ret c ; error – fall back to tape push af ; save file handle ld ix, $C000 ld bc, BANK0_SIZE pop af ; A = file handle (esxDOS preserves this) push af call esx_read pop af call esx_close

```
; Repeat for other banks...
; Bank 1
ld   a, 1
call switch_bank
ld   ix, filename_bank1
call esx_open
ret  c
; ... (same pattern) ...

ret
```

filename_bank0: db “IRONCLAW.B0”, 0 filename_bank1: db “IRONCLAW.B1”, 0 filename_bank3: db “IRONCLAW.B3”, 0 filename_bank4: db “IRONCLAW.B4”, 0 filename_bank6: db “IRONCLAW.B6”, 0

The detection code:

```z80 id:ch21_esxdos_loading_divmmc_3
; Detect esxDOS presence
; Sets carry if esxDOS is NOT available
detect_esxdos:
```

```
    ; Try to open a nonexistent file -- if RST $08 returns
    ; without crashing, esxDOS is present
    ld   a, '*'
    ld   b, FA_READ
    ld   ix, test_filename
    rst  $08
    db   F_OPEN
    jr   c, .not_present   ; carry set = open failed, but esxDOS handled it
    ; File actually opened -- close it and return success
    call esx_close
    or   a                 ; clear carry
    ret
.not_present:
    ; esxDOS returned an error -- it IS present, just file not found
    ; Distinguish from "RST $08 went to ROM and crashed"
    ; by checking if we're still running. If we're here, esxDOS is present.
    or   a                 ; clear carry = esxDOS present
    ret

test_filename:  db "IRONCLAW.B0", 0
```

In practice, the safest detection method checks for the DivMMC's identification byte at a known trap address, or uses a known-safe RST $08 call. The method above works because if esxDOS is not present, `RST $08` jumps to the ROM's error handler, which for the 128K ROM at address `$0008` is a benign return that leaves carry clear. Production code should use a more robust check; the pattern above illustrates the concept.

21.9 Loading Screen, Menu, and High Scores

Loading Screen

The loading screen is the player's first impression. It loads as `LOAD "" SCREEN$` in the BASIC loader, meaning it appears while the remaining data blocks are loading from tape. On esxDOS, the loading is fast enough that you may want to display the screen for a minimum duration:

"'z80 id:ch21_loading_screen show_loading_screen: ; Loading screen is already in screen memory ($4000) from BASIC loader ; If loading from esxDOS, load it explicitly: ld ix, filename_screen call esx_open ret c push af ld ix, $4000 ld bc, 6912 pop af push af call esx_read pop af call esx_close

```
; Minimum display time: 100 frames (2 seconds)
ld   b, 100
```

.wait: halt djnz .wait ret

filename_screen: db "IRONCLAW.SCR", 0

```
The loading screen itself is a standard Spectrum screen file: 6,144 bytes of pixel data follow
compatible art tool (ZX Paintbrush, SEViewer, or Multipaint) or convert a modern image with a
```

```
### Title Screen and Menu

The title screen state displays the game logo and animated background, then transitions to the

```z80 id:ch21_title_screen_and_menu
state_title:
    ; Animate background (e.g., scrolling starfield, colour cycling)
    call title_animate

    ; Check for keypress
    xor  a
    in   a, ($FE)          ; read all keyboard half-rows at once
    cpl                    ; invert (keys are active low)
    and  $1F               ; mask 5 key bits
    jr   z, .no_key
    ld   a, STATE_MENU
    ld   (current_state), a
.no_key:
    jp   main_loop
```

The menu offers three options: Start Game, Options, High Scores. Navigation uses up/down keys, selection uses fire/enter. The menu is a simple state machine within the `STATE_MENU` handler:

"'z80 id:ch21_title_screen_and_menu_2 menu_selection: db 0 ; 0=Start, 1=Options, 2=HiScores

state_menu: ; Draw menu (only redraw on selection change) call draw_menu

```
; Read input
call read_input
ld   a, (input_state)

; Up
bit  3, a
jr   z, .not_up
ld   a, (menu_selection)
or   a
jr   z, .not_up
dec  a
ld   (menu_selection), a
call play_menu_beep
```

.not_up:

```
; Down
ld   a, (input_state)
bit  2, a
jr   z, .not_down
ld   a, (menu_selection)
cp   2
jr   z, .not_down
inc  a
```
```

```
ld   (menu_selection), a
call play_menu_beep
```

.not_down:

```
; Fire / Enter
ld   a, (input_state)
bit  4, a
jr   z, .no_fire
ld   a, (menu_selection)
or   a
jr   nz, .not_start
; Start game
call init_game
ld   a, STATE_GAMEPLAY
ld   (current_state), a
jp   main_loop
```

.not_start: cp 1 jr nz, .not_options ; Options (toggle sound, controls, etc.) call show_options jp main_loop .not_options: ; High scores ld a, STATE_HISCORE ld (current_state), a jp main_loop

.no_fire: jp main_loop

```
### High Scores

High scores are stored in a 10-entry table in bank 2's data area:

```z80 id:ch21_high_scores
; High score entry: 3 bytes name + 3 bytes BCD score = 6 bytes
; 10 entries = 60 bytes
HISCORE_COUNT equ 10
HISCORE_SIZE  equ 6

hiscore_table:
    ; Pre-filled defaults
    db   "ACE"
    db   $00, $50, $00     ; 005000 BCD
    db   "BOB"
    db   $00, $40, $00     ; 004000
    db   "CAT"
    db   $00, $30, $00     ; 003000
    ; ... 7 more entries ...
    ds   7 * HISCORE_SIZE, 0
```

Scores use BCD (Binary Coded Decimal) – two decimal digits per byte, three bytes per score, giving a maximum of 999,999 points. BCD is preferable to binary for display because converting a 24-bit binary number to decimal on a Z80 requires expensive division. With BCD, the `DAA` instruction handles carry between digits automatically, and printing requires only masking nibbles:

“‘z80 id:ch21_high_scores_2 ; Add points to score ; DE = points to add (BCD, 2 bytes, max 9999) add_score: ld hl, player_score ld a, (hl) add a, e daa ; adjust for BCD ld (hl), a inc hl ld a, (hl) adc a, d daa ld (hl), a inc hl ld a, (hl) adc a, 0 daa ld
```

(hl), a ret

player_score: db 0, 0, 0 ; 3 bytes BCD, little-endian

```
When the game ends, the code scans the high score table to see if the player's score qualifies

On esxDOS systems, the high score table can be saved to the SD card. On tape systems, high sco

---

## 21.10 Level Loading and Decompression

When the player starts a level or completes one, the game must:

1. Page in the bank containing the level data (bank 0 for levels 1-2, bank 1 for levels 3-
5)
2. Decompress the tilemap into bank 7 (the shadow screen bank, repurposed as a data buffer dur
3. Decompress the tile graphics into a buffer in bank 2 or bank 0
4. Initialise the entity array from the level's spawn table
5. Reset the viewport to the level's start position
6. Reset the scroll engine state

```z80 id:ch21_level_loading_and
; Load and initialise level
; A = level number (0-4)
load_level:
    push af

    ; Determine which bank holds this level
    cp   2
    jr   nc, .bank1
    ; Levels 0-1: bank 0
    ld   a, 0
    call switch_bank
    pop  af
    push af
    ; Look up compressed data address
    add  a, a
    ld   l, a
    ld   h, 0
    ld   de, level_ptrs_bank0
    add  hl, de
    jr   .decompress
.bank1:
    ; Levels 2-4: bank 1
    ld   a, 1
    call switch_bank
    pop  af
    push af
    sub  2                  ; offset within bank 1
    add  a, a
```
```

```
    ld   l, a
    ld   h, 0
    ld   de, level_ptrs_bank1
    add  hl, de

.decompress:
    ; HL points to 2-byte address of compressed level data in current bank
    ld   a, (hl)
    inc  hl
    ld   h, (hl)
    ld   l, a              ; HL = compressed data source (in $C000-$FFFF)

    ; Decompress tilemap into bank 7
    ; First, save current bank and switch to bank 7
    ; BUT: bank 7 is at $4000 (shadow screen), not $C000
    ; We decompress to $C000 in a temporary bank, then copy
    ; OR: decompress directly into shadow screen at $4000

    ; Simpler approach: decompress into a buffer at $8000+ area
    ; (we have ~2KB free above our code in bank 2)
    ; For large levels, use bank 7 at $4000:
    ; Enable shadow screen banking, then write to $4000-$7FFF

    ld   de, level_buffer  ; destination in bank 2 work area
    call zx0_decompress    ; ZX0 decompressor: HL=src, DE=dest

    ; Initialise entities from spawn table
    pop  af                ; A = level number
    call init_level_entities

    ; Set viewport to level start
    ld   hl, 0
    ld   (viewport_x), hl
    ld   hl, 0
    ld   (viewport_y), hl

    ; Reset scroll state
    xor  a
    ld   (scroll_pixel_offset), a
    ld   (scroll_dirty), a

    ret
```

The choice of compressor matters here. Level data is loaded once per level (during a transition screen), so decompression speed is not critical – we can afford Exomizer's ~250 T-states per byte for the best compression ratio. But tile graphics may need to be decompressed during gameplay (if tiles are banked), so Pletter's ~69 T-states per byte is preferable.

As discussed in Chapter 14, the decompressor code itself occupies memory. ZX0 at ~70 bytes is ideal for projects where code space is tight. Ironclaw includes both a ZX0 decompressor (for level data at load time) and a Pletter decompressor (for

streamed tile data during gameplay).

21.11 Profiling with DeZog

You have written all the code. It compiles. It runs. The player walks, enemies patrol, tiles scroll, music plays. But the frame budget overflows on level 3, where six enemies and three projectiles are on screen simultaneously. The border stripe shows a red band that extends past the visible screen area. You are dropping frames.

This is where DeZog earns its place in your toolchain.

What is DeZog?

DeZog is a VS Code extension that provides a full debugging environment for Z80 programs. It connects to emulators (ZEsarUX, CSpect, or its own internal simulator) and gives you:

- Breakpoints (address, conditional, logpoints)
- Step-through execution (step into, step over, step out)
- Register watches (all Z80 registers, updated in real time)
- Memory viewer (hex dump with live updates)
- Disassembly view
- Call stack
- **T-state counter** – the profiling tool we need

The Profiling Workflow

The border stripe tells you *that* you are over budget. DeZog tells you *where*.

Step 1: Isolate the slow frame. Set a conditional breakpoint at the start of the main loop that triggers only when a "frame overflow" flag is set. Add code to set this flag when the frame takes too long:

```
z80 id:ch21_the_profiling_workflow ; At the end of the gameplay frame, before
HALT:     ; Check if we're still in the current frame     ; (a simple approach:
read the raster line via floating bus     ;  or use a frame counter incremented
by IM2)     ld   a, (frame_overflow_flag)     or   a     jr   z, .ok     ; Frame
overflowed -- set debug breakpoint trigger     nop                    ; <-- set
DeZog breakpoint here .ok:
```

Step 2: Measure subsystem costs. DeZog's T-state counter lets you measure the exact cost of any code section. Place the cursor at the start of `update_enemy_ai`, note the T-state counter, step over the call, and note the new counter value. The difference is the exact cost.

A systematic profiling pass measures each subsystem:

```
Subsystem            Measured T-states   Budget %
──────────────────────────────────────────────────

read_input                     187          0.3%
update_player_physics          743          1.0%
update_player_state            412          0.6%
```

```
update_enemy_ai             4,231          5.9%   <-- worst case
check_all_collisions        2,847          4.0%
update_projectiles            523          0.7%
scroll_viewport            12,456         17.4%   <-- expensive
render_exposed_tiles       11,892         16.6%   <-- expensive
restore_backgrounds         3,214          4.5%
draw_sprites               10,156         14.2%   <-- expensive
update_hud                  1,389          1.9%
[IM2 music interrupt]       3,102          4.3%
-------------------------------------------------
TOTAL                      51,152         71.4%
Slack                      20,528         28.6%
```

That is the average case. Now profile the worst case – level 3, six enemies on screen, player near the right edge triggering a scroll:

```
Subsystem            Measured T-states   Budget %
-------------------------------------------------
read_input                     187          0.3%
update_player_physics          743          1.0%
update_player_state            412          0.6%
update_enemy_ai              5,891          8.2%   <-- 6 enemies active
check_all_collisions         4,156          5.8%   <-- more pairs
update_projectiles           1,247          1.7%   <-- 3 projectiles
scroll_viewport             14,892         20.8%   <-- scroll + new column
render_exposed_tiles        14,456         20.2%   <-- full column render
restore_backgrounds          4,821          6.7%
draw_sprites                13,892         19.4%   <-- 10 entities
update_hud                   1,389          1.9%
[IM2 music interrupt]        3,102          4.3%
-------------------------------------------------
TOTAL                       65,188         90.9%
Slack                        6,492          9.1%
```

Only 9% slack in the worst case. That is dangerously thin. One more enemy or a complex music pattern could push you over.

Step 3: Find the bottleneck. The profiling table makes it obvious: scrolling + tile rendering consume 41% of the frame in the worst case. Sprite rendering takes 19%. Enemy AI takes 8%.

Step 4: Optimise the bottleneck. Options, roughly in order of impact:

1. **Spread the scroll cost.** Instead of rendering the full new column in one frame, render half on frame N and half on frame N+1 using the double buffer (discussed in section 21.4). This cuts the scroll spike from ~29,000 to ~15,000 T-states per frame.

2. **Use compiled sprites for the player.** The player sprite is always on screen and always rendered. Switching from OR+AND masked (Chapter 16, method 2) to compiled sprites (method 5) saves ~30% per sprite draw, but increases memory usage. For one frequently-drawn entity, the trade-off is worth it.

3. **Reduce sprite overdraw.** If two enemies overlap, you are drawing pixels that will be overwritten. Sort entities by Y coordinate (back-to-front) and skip

drawing for fully occluded sprites. This helps in the worst case when entities cluster.

4. **Tighten the AI.** Profile `run_entity_ai` for each enemy type. The Shooter's line-of-sight check (scanning tile columns for occlusion) is often the most expensive AI operation. Cache the result: only re-check line-of-sight every 8 frames instead of every 3.

After optimisation, the worst case drops to ~58,000 T-states, leaving 19% slack. That is comfortable.

DeZog Configuration for Ironclaw

DeZog connects to an emulator that supports its debug protocol. For ZX Spectrum 128K development, ZEsarUX is the recommended choice:

```
// .vscode/launch.json
{
    "version": "0.2.0",
    "configurations": [
        {
            "type": "dezog",
            "request": "launch",
            "name": "Ironclaw (ZEsarUX)",
            "remoteType": "zesarux",
            "zesarux": {
                "hostname": "localhost",
                "port": 10000
            },
            "sjasmplus": [
                {
                    "path": "src/main.a80"
                }
            ],
            "topOfStack": "0xBFFF",
            "load": "build/ironclaw.sna",
            "startAutomatically": true,
            "history": {
                "reverseDebugInstructionCount": 100000
            }
        }
    ]
}
```

The `history` setting enables reverse debugging – you can step backwards to see how you arrived at a bug. This is invaluable for tracking down collision glitches where an entity teleported through a wall three frames ago.

21.12 The Data Pipeline in Detail

Getting data from artist tools into the game is often the most underestimated part of a project. Ironclaw's pipeline converts four kinds of assets:

Tilesets (PNG to Spectrum pixel format)

The artist draws tiles in Aseprite, Photoshop, or any pixel art tool as an indexed-colour PNG. Tiles are arranged in a grid on a single sheet. The conversion script:

1. Reads the PNG, verifies it is 1-bit (black and white) or indexed with Spectrum-compatible colours
2. Slices into 8x8 or 16x16 tiles
3. Converts each tile to the Spectrum's interleaved pixel format (where row 0 is at offset 0, row 1 at offset 256, not offset 1 – matching the screen layout)
4. Optionally deduplicates identical tiles
5. Writes a binary blob and a symbol table mapping tile IDs to offsets

For attributes, each tile also carries a colour byte (INK + PAPER + BRIGHT). The script extracts this from the PNG's palette and writes a parallel attribute table.

Sprite Sheets (PNG to pre-shifted sprite data)

Sprites follow a similar pipeline, but with an additional step: pre-shifting. The conversion script:

1. Reads the PNG sprite sheet
2. Slices into individual frames
3. Generates a mask for each frame (any non-background pixel produces a 0 in the mask, background produces 1)
4. For each frame, generates 4 horizontally shifted variants (0, 2, 4, 6 pixels offset)
5. Each shifted variant is widened by one byte (a 2-byte-wide sprite becomes 3 bytes wide to hold shifted overflow)
6. Writes interleaved data+mask bytes for efficient rendering

Level Maps (Tiled JSON to binary tilemap)

Levels are designed in Tiled, a free cross-platform tilemap editor. The designer places tiles visually, adds object layers for entity spawn points and triggers, and exports as JSON or TMX.

The conversion script:

1. Reads the Tiled export
2. Extracts the tile layer as a flat array of tile indices
3. Extracts the object layer for spawn points (enemy positions, player start, item locations)
4. Generates a collision map: for each tile, looks up whether it is solid, a platform, a hazard, or empty (based on a tile properties file)
5. Writes the tilemap, collision map, and spawn table as separate binary files

Music (Vortex Tracker II to PT3)

Music is composed in Vortex Tracker II, which exports directly to `.pt3` format. The PT3 file is embedded into the bank data with `INCBIN`. The PT3 player code (widely available as open-source Z80 assembly, typically 1.5-2KB) is included in the music bank alongside the song data.

Putting It Together

The complete conversion pipeline for a level:

```
tileset.png ──→ png2tiles.py ──→ tileset.bin ──→ pletter ──→ tileset.bin.plt
                                                                    │
level1.tmx ──→ map2bin.py ──→ level1_map.bin ──→ zx0 ──→ level1_map.bin.zx0
               └─→ level1_collision.bin ──→ zx0 ──→ level1_col.bin.zx0
               └─→ level1_spawns.bin (uncompressed, small)
                                                                    │
player.png ──→ png2sprites.py ──→ player.bin (pre-shifted) ──┘
enemies.png ──→ png2sprites.py ──→ enemies.bin              ──┘
                                                                    │
level1.pt3 ──→ (direct INCBIN) ──────────────────────────────────┘
                                                                    │
sjasmplus main.a80 ──→ INCBIN all of the above ──→ ironclaw.tap
```

Every step is automated by the Makefile. The artist changes a tile, types `make`, and sees the result in the emulator.

21.13 Release Format: Building the .tap

The final deliverable is a `.tap` file. sjasmplus can generate `.tap` output directly using its `SAVETAP` directive:

“‘z80 id:ch21_release_format_building_the ; main.a80 – top-level assembly file

```
; Define the BASIC loader
DEVICE ZXSPECTRUM128

; Page in bank 2 at $8000
ORG $8000

; Include all game code
INCLUDE "defs.a80"
INCLUDE "banks.a80"
INCLUDE "render.a80"
INCLUDE "sprites.a80"
INCLUDE "entities.a80"
INCLUDE "physics.a80"
INCLUDE "collisions.a80"
INCLUDE "ai.a80"
INCLUDE "player.a80"
INCLUDE "hud.a80"
INCLUDE "menu.a80"
INCLUDE "loader.a80"
INCLUDE "music_driver.a80"
INCLUDE "sfx.a80"
INCLUDE "esxdos.a80"

; Entry point
```

entry: di ld sp, $BFFF call init_system call detect_esxdos jr c, .tape_load call load_from_esxdos jr .loaded .tape_load: call load_bank_data .loaded: call init_interrupts ei jp main_loop

```
; Bank data sections
; Each SLOT/PAGE directive places data into the correct bank
SLOT 3              ; use $C000 slot
PAGE 0              ; bank 0
ORG $C000
INCLUDE "data/bank0_levels.a80"   ; INCBIN compressed level data

PAGE 1              ; bank 1
ORG $C000
INCLUDE "data/bank1_levels.a80"

PAGE 3              ; bank 3
ORG $C000
INCLUDE "data/bank3_sprites.a80"

PAGE 4              ; bank 4
ORG $C000
INCLUDE "data/bank4_music.a80"

PAGE 6              ; bank 6
ORG $C000
INCLUDE "data/bank6_sfx.a80"

; Save as .tap with BASIC loader
SAVETAP "build/ironclaw.tap", BASIC, "Ironclaw", 10, 2
SAVETAP "build/ironclaw.tap", CODE, "Screen", $4000, 6912, $4000
SAVETAP "build/ironclaw.tap", CODE, "Code", $8000, $-$8000, $8000

; Save bank snapshots (for .sna or manual loading)
SAVESNA "build/ironclaw.sna", entry


The exact SAVETAP syntax varies by sjasmplus version. For 128K games with banked data, the cle

### Testing the Release

Before publishing, test on at least three emulators:

1. **Fuse** -- the reference Spectrum emulator, accurate timing for original hardware
2. **Unreal Speccy** -- Pentagon timing, the demoscene standard, good debugger
3. **ZEsarUX** -- supports 128K banking, esxDOS emulation, DeZog integration

And if possible, test on real hardware with a DivMMC. Emulators occasionally differ in timing

---

## 21.14 Final Polish
```

```
The difference between a working game and a finished game is polish. Here is a checklist of sm

**Screen transitions.** Do not jump between screens instantly. A simple fade-to-
black (write decreasing brightness to all attributes over 8 frames) or a wipe (clear columns f

**Death animation.** When the player dies, freeze gameplay for 15 frames, flash the player spr

**Screen shake.** When the boss hits the ground or an explosion goes off, shift the viewport b
2 pixels for 4-6 frames. On the Spectrum, you can fake this by adjusting the scroll offset wit

**Attract mode.** After 30 seconds on the title screen with no input, start a demo playback --

**Colour cycling.** Animate menu text or logo colours by cycling attributes through a palette
byte attribute cycle costs essentially zero CPU and makes static screens feel alive.

**Input debounce.** Ignore key presses that last fewer than 2 frames. Without debounce, the me

```z80 id:ch21_final_polish
; Debounced fire button
fire_held_frames:
    db   0

check_fire:
    ld   a, (input_state)
    bit  4, a
    jr   z, .released
    ; Fire is held
    ld   a, (fire_held_frames)
    inc  a
    ld   (fire_held_frames), a
    cp   1                  ; only trigger on first frame of press
    ret                     ; Z flag set if this is the first frame
.released:
    xor  a
    ld   (fire_held_frames), a
    ret                     ; Z flag clear (no fire)
```

Summary

- **Project structure matters.** Separate source files by subsystem, data files by type. Use a Makefile to automate the full pipeline from PNG/TMX to `.tap`.
- **Memory map carefully.** Code in bank 2 (fixed at `$8000`), level data in banks 0-1, sprite graphics in bank 3, music in banks 4 and 6, shadow screen in bank 7. Keep a shadow copy of port `$7FFD` – it is write-only.
- **The interrupt handler owns the music.** The IM2 handler pages in the music bank, runs the PT3 player, updates SFX, and restores the previous bank. Keep it lean – ~3,000 T-states maximum.
- **The gameplay frame budget on Pentagon is 71,680 T-states.** A typical
```

frame with scrolling, 8 sprites, and AI costs ~50,000 T-states average, ~65,000 worst case. Profile and optimise the worst case, not the average.

- **Scrolling is the most expensive single operation.** Use the combined scroll method (character-level LDIR + pixel offset) with shadow screen double-buffering. Spread the column copy across two frames when possible.
- **Run enemy AI every 2nd or 3rd frame.** Physics and collision detection run every frame; AI decisions can be amortised. This saves 2,000-3,000 T-states per frame in the worst case.
- **Use esxDOS for modern hardware.** The `RST $08` / `F_OPEN` / `F_READ` / `F_CLOSE` API is simple and fast. Detect DivMMC at startup and fall back to tape loading if absent.
- **Profile with DeZog.** The border stripe tells you that you are over budget. DeZog tells you where. Measure each subsystem, find the bottleneck, optimise it, measure again.
- **Choose the right compressor for each job.** Exomizer or ZX0 for one-time level loading (best ratio). Pletter for tile streaming during gameplay (fast decompression). See Chapter 14 for the full tradeoff analysis.
- **Polish is not optional.** Screen transitions, death animations, screen shake, input debounce, and attract mode are the difference between a tech demo and a game.
- **Test on multiple emulators and real hardware.** Fuse, Unreal Speccy, and ZEsarUX each model timing differently. DivMMC behaviour on real hardware can differ from emulated esxDOS.

Sources: World of Spectrum (ZX Spectrum 128K memory map and port $7FFD documentation); Introspec "Data Compression for Modern Z80 Coding" (Hype 2017); esxDOS API documentation (DivIDE/DivMMC wiki); DeZog VS Code Extension documentation (GitHub: maziac/DeZog); sjasmplus documentation (SAVETAP, DEVICE, SLOT, PAGE directives); Vortex Tracker II PT3 format specification; Chapters 11, 14, 15, 16, 17, 18, 19 of this book.

Chapter 22: Porting — Agon Light 2

> "The same instruction set, a completely different machine."

You have built the game. Five levels, four enemy types, a boss fight, AY music with sound effects, a loading screen, and a menu system — all running on a ZX Spectrum 128K at 3.5 MHz, in 128 kilobytes of banked memory, rendering through a ULA that has not changed since 1982. Every byte is accounted for. Every cycle is earned.

Now you are going to port it to a machine that has the same CPU instruction set, five times the clock speed, four times the memory, hardware sprites, hardware tilemap scrolling, an SD card for loading, and a 24-bit flat address space with no banking.

This should be easy.

It is not easy. It is *different* in ways that will surprise you, and the surprises teach you things about both machines that you would not learn any other way.

Same ISA, Different World

The Agon Light 2 runs a Zilog eZ80 at 18.432 MHz with 512 KB of flat RAM. The eZ80 is a direct descendant of the Z80 — it executes the entire Z80 instruction set, uses the same register names, the same flags, the same mnemonics. If you write `LD A,(HL)` on a Spectrum and `LD A,(HL)` on an Agon, the opcode is identical. The behaviour is identical. A Z80 programmer can sit down at an Agon and start writing code immediately.

But the Agon is not a fast Spectrum. It is a fundamentally different architecture wearing a familiar face. The differences fall into three categories:

What the eZ80 adds. 24-bit registers, 24-bit addressing, a 16 MB address space (of which 512 KB is populated), new instructions for 24-bit arithmetic, and a mode system (ADL vs Z80-compatible) that controls register width and address generation.

What the VDP replaces. The Spectrum's ULA — the chip that reads video memory and paints the screen — is replaced by a completely separate processor. The Agon's VDP is an ESP32 microcontroller running the FabGL graphics library. It handles display output, sprites, tilemaps, and audio. The eZ80 CPU communicates with

the VDP over a high-speed serial link, sending command sequences. There is no shared video memory. You do not write pixels to an address; you send commands to a coprocessor.

What disappears. Banked memory, contended memory, the attribute grid, the interleaved screen layout, the 6,912-byte framebuffer, the border as a timing tool, direct framebuffer access, cycle-level raster synchronization. All gone.

To port our Spectrum game, we need to understand what transfers directly, what needs rewriting, and what needs rethinking from scratch.

The Architecture at a Glance

Before diving into code, let us lay out the two machines side by side.

Feature	ZX Spectrum 128K	Agon Light 2
CPU	Z80A	eZ80 (Z80-compatible + ADL extensions)
Clock	3.5 MHz (7 MHz on turbo clones)	18.432 MHz
RAM	128 KB (8 x 16 KB banks, switched via port $7FFD)	512 KB flat (24-bit addressing)
Address space	16-bit (64 KB visible at a time)	24-bit (16 MB, 512 KB populated)
Video	ULA: 256x192, 8x8 attribute colour, direct memory-mapped	VDP (ESP32 + FabGL): multiple modes, up to 640x480, sprites, tilemaps
Framebuffer access	Direct: write to $4000–$5AFF	Indirect: send VDP commands over serial
Sprites	Software only	Hardware: up to 256, managed by VDP
Scrolling	Software only (shift entire framebuffer)	Hardware tilemap scrolling via VDP
Sound	AY-3-8910 (3 channels + noise)	VDP audio (ESP32 synthesis, multiple waveforms, ADSR)
Storage	Tape / DivMMC (esxDOS)	SD card (FAT32, MOS file API)
OS	None (bare metal) / esxDOS for file I/O	MOS (Machine Operating System)
Frame budget	~71,680 T-states (Pentagon)	~368,640 T-states (at 50 Hz)

The frame budget ratio is roughly 5:1. But this understates the real difference, because many operations that consume CPU T-states on the Spectrum — sprite rendering, screen scrolling, framebuffer management — are offloaded to the VDP on the Agon. The eZ80 CPU spends its cycles on game logic, not pixel pushing.

“‘mermaid id:ch22_the_architecture_at_a_glance graph TD subgraph ZX[“ZX Spectrum: Draw Sprite (~1200T CPU)”] direction TB ZA[“Calculate screen addressfrom (x, y) coordinates”] –> ZB[“Select pre-shifted variant(x mod 8 → shift table)”] ZB

–> ZC[“For each pixel row:AND mask with screen byteOR sprite datawrite back to framebuffer”] ZC –> ZD[“Advance to next screen row(DOWN_HL: handle interleave)”] ZD –> ZE{“16 rowsdone?”} ZE – No –> ZC ZE – Yes –> ZF[“Done — pixels invideo RAM at $4000”] end

```
subgraph AGON["Agon Light 2: Draw Sprite (~50T CPU)"]
    direction TB
    AA["Send VDU 23,27,4,N<br>(select sprite N)"] --> AB["Send VDU 23,27,13,x,y<br>(set positi
    AB --> AC["Send VDU 23,27,15<br>(update display)"]
    AC --> AD["Done — VDP renders<br>sprite in hardware"]
end

style ZC fill:#fdd,stroke:#933
style AA fill:#dfd,stroke:#393
style AB fill:#dfd,stroke:#393
style AC fill:#dfd,stroke:#393


> **The architectural shift:** On the Spectrum, the CPU _is_ the rendering engine — every pixe

---

## ADL Mode vs Z80-Compatible Mode

This is the single most important architectural concept for any Z80 programmer approaching the

The eZ80 has two operating modes:

**Z80-compatible mode (Z80 mode).** Registers are 16 bits wide. Addresses are 16 bits. The MBA
bit value, `JP (HL)` jumps to a 16-bit address (with MBASE prepended), `PUSH HL` pushes 2 byte

**ADL mode (Address Data Long).** Registers are 24 bits wide. Addresses are 24 bits. `LD HL,$0
bit value, `JP (HL)` jumps to a full 24-bit address, `PUSH HL` pushes 3 bytes onto the stack.

MOS boots the Agon in ADL mode. Your application starts in ADL mode. Most Agon software runs e
compatible mode exists, and understanding the interaction between the two is critical.

### Why You Might Use Z80 Mode

If you are porting Z80 code from the Spectrum, you might think: "just switch to Z80 mode and r
bit address calculations, your `DJNZ` loops, your `LDIR` block copies --- they all behave iden
bit addresses map to the right region of the Agon's memory.

The problem is *interacting with everything else*. MOS API calls expect ADL mode. VDP commands
bit stack frames. If you are in Z80 mode and call a MOS routine, the stack frame will be wrong
mode code pushed 2. The result is stack corruption and a crash.

### The Mode Switching Mechanism

The eZ80 provides special prefixes for switching modes within a single instruction:

| Prefix | Effect |
```

```
|--------|--------|
| `.SIS` (suffix) | Execute the following instruction in Z80 mode (short registers, short addr
| `.LIS` | Execute in: Long registers, Short addresses |
| `.SIL` | Execute in: Short registers, Long addresses |
| `.LIL` | Execute in ADL mode (long registers, long addresses) |

And for calls and jumps:

| Instruction | From mode | To mode |
|-------------|-----------|---------|
| `CALL.IS addr` | ADL | Z80 |
| `CALL.IL addr` | Z80 | ADL |

The `.IS` suffix means "Instruction Short" --- the call instruction itself uses short (16-
bit) conventions for the return address. `.IL` means "Instruction Long" --- the call pushes a
bit return address.

Here is the practical pattern for calling MOS from Z80-mode code:

```z80 id:ch22_the_mode_switching_mechanism
; In Z80-compatible mode, calling a MOS API function
; We need to switch to ADL mode for the call

    ; Method: use RST.LIL $08 (MOS API entry point)
    ; .LIL means "long instruction, long mode" ---
    ; pushes a 24-bit return address and enters ADL mode
    RST.LIL $08        ; call MOS API in ADL mode
    DB      mos_func   ; MOS function number follows
    ; MOS returns to us in Z80 mode (matching our caller)
```

MOS provides RST $08 as a unified API entry point. The `.LIL` suffix handles the mode transition cleanly. After the call, execution returns to your Z80-mode code with the correct stack state.

The Practical Rule

For porting, the cleanest approach is: **run your game logic in ADL mode and translate your Z80 code to use 24-bit conventions from the start.** Do not try to run in Z80 mode and switch back and forth for every MOS call. The mode-switching overhead and the risk of stack mismatches are not worth it.

This means your port will not be a byte-for-byte copy of the Spectrum code. It will be a *translation*. The algorithms are the same. The logic is the same. The register usage is mostly the same. But every address is 24 bits wide, every stack push is 3 bytes, and every immediate address load carries an extra byte.

The MBASE Trap

If you do use Z80 mode, MBASE determines the upper 8 bits of every memory address. On boot, MOS sets MBASE to $00, meaning Z80-mode addresses $0000—FFFF map to physical addresses $000000–$00FFFF. If your code or data lives above $00FFFF (above the first 64 KB), Z80-mode code cannot reach it without
```

changing MBASE.

This is a trap for Spectrum porters who think "I have 512 KB, I will put my level data at $080000." In Z80 mode, that address does not exist. You must either use ADL mode to access it or set MBASE to $08 (making addresses $0000 — —FFFF map to $080000–$08FFFF). But changing MBASE affects *all* memory accesses, including instruction fetches — so your code had better be in that region too, or you will jump into garbage.

The advice is simple: stay in ADL mode. Use the full 24-bit address space natively.

What Transfers Directly

Not everything changes. A surprising amount of your Spectrum game logic ports with minimal modification.

Game Logic and Entity System

The entity system from Chapter 18 — the structure arrays holding X, Y, type, state, animation frame, velocity, health, and flags — transfers almost verbatim. The main loop structure (HALT, input, update, render, repeat) is identical in concept, though the specific HALT and interrupt mechanism differs.

Here is the entity update loop on the Spectrum:

"'z80 id:ch22_game_logic_and_entity_system ; Spectrum: Update all entities ; IX points to entity array, B = entity count update_entities: ld ix,entities ld b,MAX_ENTITIES .loop: ld a,(ix+ENT_FLAGS) bit 0,a ; bit 0 = active? jr z,.skip

```
call update_entity      ; process this entity
```

.skip: ld de,ENT_SIZE ; size of one entity struct add ix,de ; advance to next entity djnz .loop ret

```
And on the Agon:

```z80 id:ch22_game_logic_and_entity_system_2
; Agon (ADL mode): Update all entities
; IX points to entity array, B = entity count
update_entities:
    ld   ix,entities        ; 24-bit address, loaded as 3 bytes
    ld   b,MAX_ENTITIES
.loop:
    ld   a,(ix+ENT_FLAGS)
    bit  0,a
    jr   z,.skip

    call update_entity

.skip:
    ld   de,ENT_SIZE        ; DE is now 24-bit; ENT_SIZE may differ
    add  ix,de              ; 24-bit add
```
```

```
    djnz .loop
    ret
```

The logic is identical. The instructions are identical. The difference is that IX, DE, and the program counter are all 24 bits wide. The assembler handles the encoding — `LD IX,entities` emits a 24-bit immediate instead of a 16-bit one. The entity struct itself might be identical, or you might widen the position fields to 24-bit for larger level maps. That is a design decision, not a porting constraint.

AABB Collision Detection

The collision code from Chapter 19 transfers directly. AABB checks use 8-bit or 16-bit comparisons — the same CP, SUB, and conditional jump instructions work identically on both machines.

```
z80 id:ch22_aabb_collision_detection ; AABB collision check: identical on both
platforms ; A = entity1.x, B = entity1.x + width ; C = entity2.x, D = entity2.x
+ width check_overlap_x:     ld   a,(ix+ENT_X)     cp   (iy+ENT_X2)       ;
entity1.x < entity2.x+width?     ret  nc                 ; no overlap     ld
a,(ix+ENT_X2)     cp   (iy+ENT_X)        ; entity1.x+width > entity2.x?     ret
c                  ; no overlap     ; overlap on X axis confirmed
```

Fixed-Point Arithmetic

All fixed-point 8.8 calculations — gravity, velocity, friction, acceleration — port without changes. The shift-and-add patterns, the 16-bit additions, the right-shift friction:

```
z80 id:ch22_fixed_point_arithmetic ; Apply gravity: velocity_y += gravity ;
Works identically on both platforms     ld   a,(ix+ENT_VY_LO)     add  a,GRAVITY_LO
ld   (ix+ENT_VY_LO),a     ld   a,(ix+ENT_VY_HI)     adc  a,GRAVITY_HI     ld
(ix+ENT_VY_HI),a
```

Byte-level arithmetic does not care whether the registers are notionally 16 or 24 bits wide. The accumulator is always 8 bits. Carry propagation works the same way.

State Machine

The game state machine (title, menu, gameplay, pause, game over) uses a jump table indexed by state number. On the Spectrum:

“‘z80 id:ch22_state_machine ; Spectrum: dispatch game state ld a,(game_state) add a,a ; multiply by 2 (16-bit pointers) ld e,a ld d,0 ld hl,state_table add hl,de ld a,(hl) inc hl ld h,(hl) ld l,a jp (hl)

state_table: dw state_title dw state_menu dw state_game dw state_pause dw state_gameover

```
On the Agon, the pointer table stores 24-bit addresses:

```z80 id:ch22_state_machine_2
; Agon (ADL mode): dispatch game state
    ld   a,(game_state)
```
```

```
    ld   l,a
    ld   h,0               ; HL = state index
    ld   e,l
    ld   d,h               ; DE = copy of state index
    add  hl,hl             ; HL = state * 2
    add  hl,de             ; HL = state * 3 (24-bit pointers)
    ld   de,state_table
    add  hl,de
    ld   hl,(hl)           ; load 24-bit pointer
    jp   (hl)

state_table:
    dl   state_title       ; DL = define long (24-bit)
    dl   state_menu
    dl   state_game
    dl   state_pause
    dl   state_gameover
```

The change: pointers are 3 bytes instead of 2, so the index multiplication changes from `*2` to `*3`, and the table uses `DL` (define long) instead of `DW` (define word). The logic is otherwise identical.

What Needs Rewriting

Rendering: From Framebuffer to VDP Commands

This is the largest single change in the port. On the Spectrum, rendering means writing bytes to video memory addresses. The entire rendering pipeline — sprite drawing, screen clearing, tile painting, scrolling — is CPU code that manipulates memory at $4000–$5AFF.

On the Agon, rendering means sending VDP command sequences. The VDP understands a protocol based on VDU byte streams (the same VDU command system used by BBC BASIC, extended with Agon-specific commands). You send a sequence of bytes to the VDP through MOS, and the ESP32 processes them.

Sprites

On the Spectrum (from Chapter 16), drawing a 16x16 masked sprite costs roughly 1,200 T-states of CPU time — reading mask bytes, ANDing with the screen, ORing the sprite data, writing back. You do this for every sprite, every frame.

On the Agon, you upload the sprite bitmap *once*, and then move it by sending a position update:

“‘z80 id:ch22_rendering_from_framebuffer_to ; Agon: Create and position a hardware sprite ; Step 1: Upload sprite bitmap (done once at init) ; VDU 23, 27, 4, spriteNum ; select sprite ; VDU 23, 27, 0, w, h ; set dimensions ; followed by pixel data

; Step 2: Move sprite (done every frame) ; VDU 23, 27, 4, spriteNum ; select sprite ; VDU 23, 27, 13, x.lo, x.hi, y.lo, y.hi ; set position

move_sprite: ; Send VDU command to move sprite ld a,23 rst $10 ; MOS: output byte to VDP ld a,27 rst $10 ld a,4 ; command: select sprite rst $10 ld a,(sprite_num) rst $10

```
ld   a,23
rst  $10
ld   a,27
rst  $10
ld   a,13              ; command: move sprite to
rst  $10

ld   a,(sprite_x)      ; X low byte
rst  $10
ld   a,(sprite_x+1)    ; X high byte
rst  $10
ld   a,(sprite_y)      ; Y low byte
rst  $10
ld   a,(sprite_y+1)    ; Y high byte
rst  $10

; VDU 23, 27, 15        ; show sprite (update display)
ld   a,23
rst  $10
ld   a,27
rst  $10
ld   a,15
rst  $10
ret


Each `RST $10` sends one byte to the VDP through MOS. The total CPU cost of moving a sprite is
states per RST call = ~390 T-states. Compare that to the Spectrum's ~1,200 T-
states for a full masked sprite draw. And the Agon version does not need background save/resto

The trade-off: latency. The VDP processes commands asynchronously. Between sending the "move s

#### Tilemap Scrolling

On the Spectrum, horizontal scrolling means shifting every byte of video memory left or right

On the Agon, the VDP supports hardware tilemaps:

```z80 id:ch22_rendering_from_framebuffer_to_2
; Agon: Set up a tilemap (done once)
; VDU 23, 27, 20, tileWidth, tileHeight
; VDU 23, 27, 21, mapWidth.lo, mapWidth.hi, mapHeight.lo, mapHeight.hi

; Scroll the tilemap (every frame)
; VDU 23, 27, 24, offsetX.lo, offsetX.hi, offsetY.lo, offsetY.hi

scroll_tilemap:
    ld   a,23
```
```

```
    rst  $10
    ld   a,27
    rst  $10
    ld   a,24              ; command: set scroll offset
    rst  $10

    ld   hl,(scroll_x)
    ld   a,l
    rst  $10               ; offsetX low
    ld   a,h
    rst  $10               ; offsetX high
    ld   hl,(scroll_y)
    ld   a,l
    rst  $10               ; offsetY low
    ld   a,h
    rst  $10               ; offsetY high
    ret
```

Eight bytes sent. Perhaps 240 T-states of CPU time. On the Spectrum, a full-screen horizontal pixel scroll costs tens of thousands of T-states. The Agon does it in hardware for nearly free.

But you must set up the tilemap first: upload tile definitions, define the map dimensions, populate the map with tile indices. This is a one-time cost at level load, not a per-frame cost. On the Spectrum, your tile data lives in banked RAM and is rendered into the framebuffer by your own code. On the Agon, tile data lives in VDP memory and is rendered by the ESP32. Your role changes from "graphics engine programmer" to "VDP command sequencer."

Screen Layout

The entire nightmare of the Spectrum's interleaved screen layout — the split addressing, the DOWN_HL routines, the careful calculations to convert (x, y) coordinates to memory addresses — vanishes. The Agon's VDP works in screen coordinates. You say "draw at (100, 50)" and the VDP handles the rest.

This means the DOWN_HL routine from Chapter 2, the screen address lookup tables, the attribute address calculations — none of it ports. It is simply deleted. The equivalent operation on the Agon is "send a coordinate pair to the VDP."

What Needs Rethinking

Some Spectrum patterns are so deeply embedded in the game architecture that you cannot just rewrite the rendering layer. The *design* needs to change.

Memory Architecture

On the Spectrum, you carefully planned which data goes in which bank:

- Banks 0–3: level data, tilesets, sprite graphics
- Banks 4–6: music patterns, sound effects, lookup tables

- Bank 7: shadow screen for double buffering

Every bank switch costs a port write and limits what code can see what data. The game architecture is shaped by the 16 KB window into a 128 KB space.

On the Agon, all 512 KB is visible simultaneously. There is no banking. There is no shadow screen trick (the VDP handles double buffering internally). You can have your entire game — all five levels, all tilesets, all sprites, all music — resident in memory at once. Level transitions do not require loading from tape or disk; you just point to a different region of RAM.

This simplifies development, but it also removes a constraint that forced good architecture. On the Spectrum, you were forced to think about data locality, about what needed to be co-resident, about load sequences. On the Agon, you can be sloppy. Do not be sloppy. The Agon has 512 KB, not infinity. A well-organized memory map is still a virtue.

Typical Agon memory layout for the ported game:

```
$000000 - $00FFFF   MOS and system (reserved)
$040000 - $04FFFF   Game code (~64 KB)
$050000 - $06FFFF   Level data, all 5 levels (~128 KB)
$070000 - $07FFFF   Music and SFX data (~64 KB)
$080000 - $0FFFFF   Free / working buffers
```

Everything is addressable with a single `LD HL,$070000` — no bank switching, no port writes.

Loading

On the Spectrum, loading from tape is a minutes-long process with a distinctive audio signature. Even with DivMMC and esxDOS, file access is a sequence of RST $08 calls:

“‘z80 id:ch22_loading ; Spectrum + esxDOS: load a file ld a,’*’ ; current drive ld ix,filename ld b,$01 ; read-only rst $08 ; esxDOS call DB $9A ; F_OPEN ; A = file handle

```
ld   ix,buffer
ld   bc,size
rst  $08
DB   $9D              ; F_READ
; Data loaded

rst  $08
DB   $9B              ; F_CLOSE


On the Agon, MOS provides a file API that reads directly from the SD card:

```z80 id:ch22_loading_2
; Agon: load a file using MOS API
    ld   hl,filename       ; 24-bit pointer to filename string
    ld   de,buffer         ; 24-bit pointer to destination
    ld   bc,size           ; 24-bit max size
    ld   a,mos_fopen       ; MOS file open function
```
```

```
    rst  $08               ; MOS API call
    ; A = file handle

    ld   a,mos_fread       ; MOS file read function
    rst  $08
    ; Data loaded

    ld   a,mos_fclose
    rst  $08
```

The pattern is similar — open, read, close — but the Agon reads from FAT32 on an SD card, which is fast enough that you can load level data between scenes without a visible delay. No loading screens needed. No tape-loading routines. No block-loading optimization.

Sound

The Spectrum's AY-3-8910 is programmed by writing directly to hardware registers through I/O ports. Every note, every envelope change, every noise burst is a specific register value written at a specific time.

The Agon's audio is handled by the VDP. You send sound commands over the same serial link used for graphics:

"'z80 id:ch22_sound ; Agon: play a note ; VDU 23, 0, 197, channel, volume, freq.lo, freq.hi, duration.lo, duration.hi

play_note: ld a,23 rst $10 xor a ; 0 rst $10 ld a,197 ; sound command rst $10 ld a,(channel) rst $10 ld a,(volume) rst $10 ld hl,(frequency) ld a,l rst $10 ld a,h rst $10 ld hl,(duration) ld a,l rst $10 ld a,h rst $10 ret

```
The Agon's sound system supports multiple waveforms (sine, square, triangle, sawtooth, noise)
channel ADSR envelopes --- features the AY does not have. But the programming model is complet
level note commands. Your AY music player --- the IM2 interrupt handler that reads pattern dat

One approach: write a thin abstraction layer that both platforms share.

```z80 id:ch22_sound_2
; Abstract sound interface
; Spectrum implementation:
sound_play_note:
    ; A = channel, B = note, C = instrument
    ; ... look up AY register values, write to ports $FFFD/$BFFD
    ret

; Agon implementation:
sound_play_note:
    ; A = channel, B = note, C = instrument
    ; ... convert to VDP sound command, send via RST $10
    ret
```

Same call signature. Different internals. The game code calls `sound_play_note` without knowing which platform it is running on.
```

Input

The Spectrum reads keyboard state by probing port $FE with half-row addresses in the accumulator. Kempston joystick is read from port $1F. These are raw I/O port reads with specific bit patterns.

The Agon reads keyboard state through MOS:

“‘z80 id:ch22_input ; Spectrum: read keyboard ld a,$FD; half - row : QWERTina, $(FE) bit 0,a ; bit 0 = Q ; ...

; Agon: read keyboard via MOS ld a,mos_getkey ; MOS: get key state rst $08 ; A = key code, or 0 if no key pressed

```
The Spectrum gives you a bitmask of simultaneously pressed keys, ideal for game input (you can
based: it gives you the most recent key pressed. For game input with simultaneous key detectio
```

```z80 id:ch22_input_2
; Agon: read simultaneous keys from keyboard map
    ld   a,(mos_keymap+KEY_LEFT)   ; 1 if left arrow held, 0 if not
    or   a
    jr   z,.no_left
    ; move player left
.no_left:
    ld   a,(mos_keymap+KEY_RIGHT)
    or   a
    jr   z,.no_right
    ; move player right
.no_right:
```

This is functionally equivalent to the Spectrum’s bitmask approach, just organized differently. The port is straightforward: replace port reads with memory reads from the keyboard map.

---

eZ80 at 18 MHz: What Still Matters, What Does Not

The eZ80 is roughly five times faster than the Z80A in raw clock speed. But many eZ80 instructions also execute in fewer clock cycles than their Z80 equivalents — single-byte register instructions often complete in 1 cycle instead of 4. The effective speedup for typical code is somewhere between 5x and 20x depending on the instruction mix.

This changes the optimization calculus dramatically.

What Still Matters: Inner Loop Efficiency

Even at 18 MHz with 368,000 T-states per frame, the inner loop still matters for CPU-intensive operations. If you are doing collision checks against a tilemap, iterating over 200 entities, or processing AI state machines for dozens of enemies, the per-iteration cost of your hot loop adds up.

The core Z80 optimization techniques — keeping values in registers instead of memory, using `INC L` instead of `INC HL` where possible, avoiding IX/IY indexed instructions for hot paths (they carry a 2-cycle prefix penalty on eZ80, just as they carry a 4-T-state penalty on Z80) — still produce measurable improvements.

“‘z80 id:ch22_what_still_matters_inner_loop ; Tight entity scan: same optimization principles on both platforms ; Prefer: register-to-register ops, direct addressing, DJNZ ; Avoid: IX-indexed loads in hot inner loops when possible

scan_entities_fast: ld hl,entity_flags ; pointer to flags array ld b,MAX_ENTITIES .loop: ld a,(hl) bit 0,a call nz,process_entity inc hl ; next entity flag (assume 1-byte stride) djnz .loop ret

```
This pattern --- minimal memory access, tight register usage, `DJNZ` loop control --- is optim

### What Becomes Irrelevant: Memory Conservation Tricks

On the Spectrum, memory pressure drives much of the engineering. Pre-shifted sprites store 4 o
8x the memory, as a speed/memory trade-off. Compression is mandatory for fitting demo data int

On the Agon, with 512 KB of flat RAM and SD card storage for anything that does not need to be
shifted copies of every sprite without worrying about memory. You can keep all lookup tables a

This does not mean you should waste memory. But it means that optimization decisions driven by
byte sine table or a 128-byte one?" --- become irrelevant. Use 256. Use 1,024 if the precision

### What Becomes Irrelevant: Self-Modifying Code

On the Spectrum, self-modifying code (SMC) is a standard optimization technique. You write ins

```z80 id:ch22_what_becomes_irrelevant_self
; Spectrum: self-modifying code for speed
    ld   a,0               ; operand patched at runtime
    ; ... (the $00 after LD A is overwritten with the real value)
```

On the eZ80, SMC still works (the eZ80 does not have an instruction cache that would invalidate), but the motivation is weaker. The extra cost of a memory load is smaller relative to the total frame budget. More importantly, MOS maps some memory regions as read-only, and certain Agon firmware versions may restrict execution of modified code depending on memory region. SMC is not prohibited on the Agon, but it is rarely necessary and can cause subtle problems.

What Becomes Irrelevant: Stack Tricks for Rendering

On the Spectrum, abusing the stack pointer for fast screen fills (DI, set SP to screen address, PUSH data repeatedly) is a classic trick because PUSH writes 2 bytes in 11 T-states — faster than any other write mechanism. On the Agon, you are not writing to a framebuffer at all. The VDP handles rendering. Stack tricks for display are meaningless.

```

The Comparison Table

Here is the side-by-side comparison of the same game on both platforms. These numbers are representative of the platformer built in Chapters 21–22.

Metric	ZX Spectrum 128K	Agon Light 2
Game code size	~12 KB	~14 KB
Rendering code	~5 KB (sprite engine, scroll, screen mgmt)	~2 KB (VDP command sequences)
Total code	~17 KB	~16 KB
Level data (all 5)	~40 KB (compressed, loaded per-level)	~60 KB (uncompressed, all resident)
Sprite/tile graphics	~20 KB (packed, 1bpp + masks)	~80 KB (8bpp RGBA, uploaded to VDP)
Music + SFX	~16 KB (PT3 + SFX tables)	~20 KB (converted format + waveform data)
Total data	~76 KB (fits in 128 KB with banking)	~160 KB (fits in 512 KB easily)
Compression needed?	Yes, mandatory	No, optional
Sprite draw cost	~1,200 T/sprite (software)	~400 T/sprite (VDP commands)
Scroll cost per frame	~15,000–30,000 T (software shift)	~240 T (VDP offset command)
Frame budget	~71,680 T	~368,640 T
Achievable fps	25–50 (depends on entity count)	60 (VDP-limited, not CPU-limited)
Dev complexity	High (memory banking, screen layout, rendering engine)	Medium (VDP protocol, MOS API, ADL mode)
Visual result	Monochrome or 2-colour-per-cell sprites, attribute colour, 256x192	Full colour sprites, smooth scrolling, 320x240 or higher

The code size difference is instructive. On the Spectrum, the *rendering engine* is a substantial fraction of the codebase. On the Agon, VDP commands replace most of that code. But the Spectrum game is smaller in total data because everything is compressed and packed tightly. The Agon version uses more memory for richer assets (8-bit-per-pixel sprites instead of 1-bit-per-pixel with masks).

The frame budget comparison is the most striking number. The Agon game is *CPU-idle most of the frame*. After processing game logic, sending sprite updates, and handling input, the eZ80 has nothing to do until the next frame. On the Spectrum, you are fighting for every cycle to fit everything into the budget.

The Porting Process: Step by Step

Here is the practical sequence for porting the Chapter 21 game to the Agon.

Step 1: Set Up the Agon Project

Create a new project with the Agon toolchain. You will need:

- An eZ80 assembler (ez80asm or the Zilog ZDS tools)
- The MOS API header file (mos_api.inc) defining function numbers and constants
- A way to transfer the binary to the Agon (SD card or serial upload)

Your entry point is different from the Spectrum. Instead of `ORG $8000` with a raw JP to your start address, the Agon loads executables at $040000 and MOS passes control to that address:

“‘z80 id:ch22_step_1_set_up_the_agon ; Agon: application entry point .ASSUME ADL=1 ; we are in ADL mode ORG $040000

```
JP   main              ; standard entry
```

main: ; Your game starts here ; … initialize VDP, load assets, enter game loop

```
### Step 2: Replace the Rendering Layer

This is the bulk of the work. Delete the Spectrum rendering code and replace it with VDP comma

1. **Upload sprite bitmaps to VDP.** Convert your 1bpp sprite data to the VDP's bitmap format

2. **Upload tileset to VDP.** Convert your 8x8 tiles from Spectrum attribute format to the VDP

3. **Replace the sprite draw routine** with VDU 23,27 sprite position commands (as shown earli

4. **Replace the scroll routine** with VDU 23,27 tilemap scroll offset commands.

5. **Delete the screen address calculation code**, the DOWN_HL routine, the attribute address

### Step 3: Translate the Game Logic

Walk through the game logic code (entity update, collision detection, physics, AI, state machi

- Change all `DW` (define word) to `DL` (define long) for address tables.
- Change pointer arithmetic from 16-bit to 24-bit where addresses are involved.
- Verify that `PUSH`/`POP` pairs are balanced --- each push is now 3 bytes, not 2.
- Check that `LDIR` and `LDDR` block copy counts are correct (BC is 24-bit in ADL mode; if you

### Step 4: Rewrite Sound

Write a new music driver that reads your pattern data and emits VDP sound commands instead of

### Step 5: Rewrite Input

Replace port reads with MOS keyboard map reads. The keymap approach is simple and provides sim

### Step 6: Rewrite Loading

Replace esxDOS file operations with MOS file API calls. The pattern is similar; only the funct
```

```
### Step 7: Test and Tune

Run the game. Verify sprite positions, collision boxes, scrolling speed, sound timing. The VDP

---

## What Each Platform Forces You To Do Better

The Spectrum teaches cycle-level efficiency and creative constraint-solving: you learn to coun
states, exploit memory layout, and invent techniques like multicolour and attribute-
only effects that exist only because of the hardware's limitations. The Agon teaches system ar

---

## A Note on eZ80 Instructions

The eZ80 adds several instructions that Z80 programmers will appreciate. The most useful for g

**LEA (Load Effective Address).** Compute an address from a base register plus an 8-
bit signed displacement, without modifying the base register:

```z80 id:ch22_a_note_on_ez80_instructions
; eZ80: LEA IX, IY + offset
; Compute IX = IY + displacement without changing IY
    LEA  IX,IY+ENT_SIZE    ; IX points to next entity
```

On the Z80, this requires `PUSH IY` / `POP IX` / `LD DE,ENT_SIZE` / `ADD IX,DE` — four instructions and 40+ T-states. LEA does it in one instruction.

TST (Test Immediate). AND the accumulator with an immediate value and set flags, without modifying A:

```
z80 id:ch22_a_note_on_ez80_instructions_2 ; eZ80: TST A, mask ; Test bits without
destroying A     TST  A,$80             ; test sign bit     jr   nz,.negative
; branch if bit 7 set, A unchanged
```

On the Z80, you would need `BIT 7,A` (which does not work with arbitrary masks) or `PUSH AF` / `AND mask` / `POP AF` (expensive).

MLT (Multiply). 8x8 unsigned multiply, result in a 16-bit register pair:

```
z80 id:ch22_a_note_on_ez80_instructions_3 ; eZ80: MLT BC ; B * C -> BC (16-bit
result)     ld   b,sprite_width     ld   c,frame_number     mlt  bc               ;
BC = B * C
```

On the Z80, multiplication requires a loop or a lookup table. MLT is a single instruction. For game logic — computing sprite offsets, tile map indices, animation frame positions — this is a substantial simplification.

```

Summary

- The Agon Light 2 runs the same Z80 instruction set as the Spectrum, but with 24-bit addressing (ADL mode), 512 KB flat RAM, hardware sprites and tilemaps via the VDP coprocessor, and a ~5x larger frame budget.
- **ADL mode** is the native mode. Run your game in ADL mode. Avoid Z80-compatible mode for anything other than running legacy code that cannot be converted. Mode switching via `.LIL`/`.SIS` suffixes is available but adds complexity and risk.
- **Game logic ports directly**: entity systems, collision detection, fixed-point physics, state machines, and AI all transfer with minimal changes (mainly widening pointers from 16-bit to 24-bit).
- **Rendering must be rewritten**: the Spectrum's direct framebuffer access is replaced by VDP command sequences for sprites, tilemaps, and scrolling. CPU rendering cost drops dramatically, but you now manage an asynchronous command pipeline.
- **Sound must be rewritten**: AY register writes are replaced by VDP sound commands. The pattern data can remain the same; only the output driver changes.
- **Memory architecture simplifies**: no banking, no shadow screen tricks, no compression mandated by scarcity. All assets can be resident simultaneously.
- **Spectrum tricks that become irrelevant on Agon**: self-modifying code for speed, stack-pointer rendering, pre-shifted sprite copies for memory/speed trade-offs, interleaved screen address calculations, attribute-based visual effects.
- **Spectrum tricks that still matter on Agon**: tight inner loops, register-efficient code, data-oriented struct layout, precomputed lookup tables.
- **Each platform teaches different skills**: the Spectrum teaches cycle-level efficiency and creative constraint-solving; the Agon teaches system architecture, coprocessor communication, and data pipeline management.
- **The eZ80 adds useful instructions**: LEA for address computation, MLT for hardware multiply, TST for non-destructive bit testing — all simplifications that eliminate multi-instruction Z80 patterns.

Sources: Zilog eZ80 CPU User Manual (UM0077); Agon Light 2 Official Documentation, The Byte Attic; Dean Belfield, "Agon Light — Programming Guide" (breakintoprogram.co.uk); FabGL Library Documentation (fabgl.com); Agon MOS API Documentation (github.com/AgonConsole8/agon-docs); Chapters 15–21 of this book.

Chapter 23: AI-Assisted Z80 Development

"Z80 they still don't know." – Introspec (spke), Life on Mars, 2024

This book was partially written with AI assistance. The chapter you are reading was drafted by Claude Code. The assembler used to build the examples – MinZ's `mza` – was built with AI assistance. The "Antique Toy" companion demo that this book documents was coded in a feedback loop between a human and an AI agent. If that makes you uncomfortable, good. That discomfort is worth examining.

This is the most self-aware chapter in the book. We are going to look honestly at what AI assistance means for Z80 development in 2026 – where it genuinely helps, where it confidently fails, and where the answer is a frustrating "it depends." We will do this with real examples, real code, and real failure cases, because the demoscene has never had patience for hype.

23.1 The Historical Parallel: HiSoft C on ZX Spectrum

Before we talk about AI, let us talk about another attempt to bring higher-level tools to the ZX Spectrum.

In 1998, *Spectrum Expert* #02 – the same issue where Dark and STS published their midpoint 3D method (Chapter 5) – reviewed the HiSoft C compiler for ZX Spectrum. The verdict was mixed. The compiler produced code that ran "10–15x faster than BASIC." It supported 33 reserved keywords, offered a stdio.lib that provided graphics at BASIC capability levels, and included `gam128.h` for 128K memory bank access.

But it had no floating-point support.

Think about that for a moment. A C compiler. On a machine where floating-point is already handled by the ROM's RST $28 calculator, sitting there in 16K of free code. And the compiler could not use it.

The conclusion from the *Spectrum Expert* reviewer was precise: "useful for speed-critical work where float isn't needed." A tool with clear strengths and hard limits, evaluated honestly.

HiSoft Pascal HP4D told a similar story. The compiler occupied 12K, leaving roughly 21K for programs. It supported real types and trig functions – SIN, COS, SQRT – and was "suitable for data processing and computational mathematics." But 21K

for your program, on a machine where a single uncompressed screen takes 6,912 bytes, means you are writing small programs or nothing.

Higher-level languages on constrained hardware have always been a compromise. They accelerate certain tasks enormously. They make other tasks impossible. The question was never "is HiSoft C good or bad?" but "what is it good *for*, and what should you still write in assembly?"

AI-assisted Z80 development is the same kind of compromise. Different shape, same question.

23.2 The Claude Code Feedback Loop

Here is how AI-assisted Z80 development actually works in practice. It is not magic. It is a loop.

The Loop

```
prompt --> code --> assemble --> error? --> fix --> assemble --> run --> wrong?
 ↪  --> fix --> run --> correct
  ^
 ↪  |

 ↪  +-------------------------------------------------------------------------------------
```

"'mermaid id:ch23_the_loop_2 graph LR A["Prompt(describe what you want)"] –> B["Generate Code(AI writes Z80 asm)"] B –> C["Assemble(sjasmplus / mza)"] C –> D{"Errors?"} D – Yes –> E["Paste errorsback to AI"] E –> B D – No –> F["Run in mzx(–screenshot)"] F –> G{"Outputcorrect?"} G – No –> H["Describe what's wrong(or paste DeZog state)"] H –> B G – Yes –> I{"Optimise?"} I – Yes –> J["Profile with DeZog(measure T-states)"] J –> A I – No –> K["Done!"]

```
style A fill:#ffd,stroke:#993
style B fill:#ddf,stroke:#339
style K fill:#dfd,stroke:#393
style D fill:#fee,stroke:#933
style G fill:#fee,stroke:#933


You describe what you want. The AI generates Z80 assembly. You assemble it. It fails -- wrong

This loop is not fundamentally different from how a human programmer works, except for one cri

### A Concrete Example

Suppose you ask Claude Code to write a routine that fills the ZX Spectrum attribute area with

```z80 id:ch23_a_concrete_example
; Fill attributes with diagonal stripes
; Pattern: alternating INK colours along diagonals
    ORG $8000
```
```

```
fill_diagonal:
    ld   hl, $5800          ; attribute memory start
    ld   b, 24              ; 24 rows

.row_loop:
    ld   c, 32              ; 32 columns per row
    ld   a, 24
    sub  b                  ; A = row index (0-23)

.col_loop:
    push af
    add  a, c               ; A = row + column
    and  7                  ; clamp to 0-7 (INK colours)
    or   $40                ; set BRIGHT bit
    ld   (hl), a            ; write attribute
    inc  hl
    pop  af
    dec  c
    jr   nz, .col_loop

    djnz .row_loop
    ret
```

This is reasonable code. It assembles. It runs. But there are problems. The column counter `C` counts down from 32, not up from 0, so the diagonal goes the wrong direction. The `or $40` sets bit 6 (BRIGHT), which is correct, but the visual effect has diagonals running top-right to bottom-left instead of the more natural top-left to bottom-right. And the PUSH/POP AF in the inner loop costs 22 T-states per attribute byte – unnecessary overhead if you restructure the calculation.

None of these are *wrong* in the sense that the code crashes. They are *wrong* in the sense that a Z80 programmer would not write the code this way. A human who has filled attributes hundreds of times would compute the diagonal index differently, avoid the PUSH/POP, and get the direction right on the first attempt because the pattern of row + column is second nature.

Here is the version you arrive at after two iterations:

“‘z80 id:ch23_a_concrete_example_2 fill_diagonal: ld hl, $5800 ld d, 0 ; row index

.row_loop: ld e, 0 ; column index ld b, 32

.col_loop: ld a, d add a, e ; diagonal = row + col and 7 or $40 ; BRIGHT + INK colour ld (hl), a inc hl inc e djnz .col_loop

```
inc  d
ld   a, d
cp   24
jr   nz, .row_loop
ret
```

```
Cleaner. No PUSH/POP. Diagonals run the right direction. The inner loop costs 4 + 4 + 7 + 4 +
states per byte -- not brilliant, but functional for a fill routine that runs once.
```

The point is not that the AI wrote bad code. The point is that the *loop* -- prompt, generate,

What Makes the Loop Fast

The loop is faster with AI than without for specific categories of work:

Boilerplate. The ORG directive, the HALT loop, the border colour timing harness from Chapt
50 lines. The AI generates these correctly and instantly. A human types them from memory. The

Iteration on a known pattern. "Now make the diagonal go the other direction." "Add a frame
BRIGHT." Each iteration is a small delta on existing code. The AI applies the delta faster tha

Test harness generation. "Write a test that fills memory at $C000 with values 0-
255, calls the multiply routine at $8000, and checks the results against a table." The AI gene

Documentation and comments. "Add cycle counts to every instruction in this inner loop." Th

What Makes the Loop Slow

Novel algorithms. When you ask for something the AI has not seen -- a new unrolling strate
looking code that is often subtly wrong. Worse, it is wrong in ways that compile and run but p
generated code than you would have spent writing it yourself.

Cycle counting under pressure. The AI can count cycles for isolated instructions. But when
taken costs, and must fit within a budget of 2,340 T-states (one scanline minus a few instruct
states" when the actual cost depends on branch probabilities and memory alignment. This is whe

Creative effect design. "Design a visual effect that looks good and fits in 8,000 T-
states" is a question the AI cannot answer. It can implement an effect you describe. It cannot

---

23.3 DeZog Integration: The Other Half of the Loop

If the AI generates the code, DeZog tells you whether it works.

DeZog is a VS Code extension that provides a Z80 debugger interface. It connects to emulators

The AI + DeZog Workflow

The most productive workflow for AI-assisted Z80 development combines Claude Code with DeZog i

1. **Claude Code generates a routine** -- say, an 8x8 multiply.
2. **You assemble it** with `mza` and load it into the DeZog-connected emulator.
3. **You set a breakpoint** at the entry point and step through.
4. **You watch the registers** at each step. Does A contain the right intermediate value after
5. **You spot a divergence** -- the high byte of the result is wrong. You take a screenshot of
6. **You paste the divergence back to Claude Code** -- "After 6 iterations of the shift loop,
7. **Claude Code identifies the bug** -- usually a missing shift, a wrong register choice, or
by-one in the loop count.
8. **You fix, reassemble, retest.**

This workflow is powerful because it gives the AI what it lacks: ground truth. The AI is good

Memory Inspection for Data-Heavy Code

For routines that manipulate memory -- screen fills, table generation, buffer operations -- De

This is especially valuable for AI-generated lookup tables. Claude Code can generate a routine
byte sine table using the parabolic approximation from Chapter 4. The routine will usually *al
by-one in the index that shifts the entire table by one position, or a sign error that inverts
good values.

What DeZog Cannot Do (Yet)

DeZog does not currently integrate with AI agents programmatically. You, the human, are the br
defined problems. For creative and architectural work, the human remains in the loop.

23.4 When AI Helps, When It Does Not

Let us be specific. Not "AI is good at some things" -- specific categories with specific asses

AI Helps: High Confidence

Instruction encoding and cycle counts. The AI has the Z80 instruction set memorised: opcod
state costs. `DJNZ` taken = 13T, not taken = 8T. `LDIR` per byte = 21T except last = 16T. It g

Boilerplate and scaffolding. ORG directives, HALT loops, AY register writes, screen clear

Dialect translation and code explanation. Converting between sjasmplus, mza, and z80asm sy

AI Helps: Medium Confidence

Standard algorithms. Shift-and-add multiply, restoring division, Bresenham line drawing, L
based scrolling. The AI generates working implementations of these, but they are usually textb
15% more speed through register allocation tricks, flag exploitation, and unrolling that the A

Memory layout and addressing. "Set up a 256-byte aligned table at $xx00" or "calculate the
boundary crossing wrong in the pixel memory interleave.

Simple self-modifying code. Patching an immediate operand, changing a jump target, swappin
modification -- where the modified code's behaviour depends on multiple patches interacting --

AI Does Not Help: Low Confidence

Novel inner loop optimisation. This is the big one. When you need to shave 3 T-
states off an inner loop that runs 6,144 times per frame -- when 3 T-states is the difference

Introspec's `ld a,(hl) : inc l : dec h : add a : add a : add (hl)` rotozoomer inner loop from
states for 4 chunky pixel pairs. The genius is in the choice to use `inc l` instead of `inc hl

states, 6 for the pair) and to exploit the fact that `add a` (a doubling) is 4T while `sla a`
decisions that accumulate into the difference between a demo that runs and a demo that does no

Contended memory timing. The delay pattern on original Spectrums (6, 5, 4, 3, 2, 1, 0, 0 e
states per 8-T-state period) interacts with instruction timing in ways the AI cannot reliably

Flag-based tricks and aesthetic judgement. The AI knows that `ADD A,A` sets carry from bit
wave.

23.5 Case Study: Building MinZ

MinZ is a programming language for Z80 and eZ80 systems, built by Alice with substantial AI as
2026. It compiles modern, readable code to efficient Z80 assembly. The project is real, open-
source, and at version 0.18.0 as of this writing.

MinZ is relevant to this chapter for two reasons. First, it is a case study in AI-
assisted development of Z80-targeting tools. Second, it is itself an example of the HiSoft C p
level language on constrained hardware, with familiar strengths and limits.

What MinZ Is

MinZ provides typed variables (`u8`, `u16`, `i8`, `i16`, `bool`), functions with multiple retu

The toolchain includes four standalone tools:

- **mza** — Z80 assembler with macros, multiple output formats (.sna, .tap, .com, .rom, .bin),
platform targets
- **mzx** — ZX Spectrum emulator with headless CLI mode, automated screenshots, keystroke inje
precise capture
- **mzd** — Z80 disassembler with IDA-like recursive descent analysis, cross-
references, T-state counting, and reassemblable output
- **MinZ compiler** — compiles MinZ source to Z80 assembly via mza

A MinZ program looks like this:

```minz
import stdlib.graphics.screen;
import stdlib.input.keyboard;
import stdlib.time.delay;

fun main() -> void {
    clear_screen();
    draw_circle(128, 96, 50);

    loop {
        wait_frame();
        let dx = get_key_dx();
        // Move sprite based on input...
    }
```

```
}
```

This compiles to Z80 assembly, assembles to a binary, and runs on real or emulated hardware. The self-contained toolchain – compiler, assembler, emulator, disassembler – means no external dependencies.

Where AI Helped Build MinZ

The compiler itself. MinZ's compiler is written in Go (~90,000 lines). The bulk of the code generation – translating MinZ's intermediate representation to Z80 assembly – was written in an AI-assisted loop. The pattern: describe the semantics of a language feature, generate the code generator, test against the emulator, fix discrepancies. For standard features like arithmetic expressions, function calls, and control flow, this loop converged quickly. Claude Code generated correct code generators for `if/else` and `while` loops on the first or second attempt.

The assembler. `mza`, the MinZ Z80 assembler, was built with AI assistance. It supports the full Z80 instruction set, macros, multiple output formats, and two-pass assembly. The instruction encoding table – which maps mnemonics to opcodes, handling all the Z80's irregular prefix byte patterns (CB, DD, ED, FD) – was generated by the AI and verified against the Z80 data sheet. This is exactly the kind of systematic, table-driven code that AI handles well.

The emulator. `mzx` achieves 100% Z80 instruction coverage, including all undocumented opcodes (ED prefix NOPs, DDCB/FDCB indexed bit operations). The AI generated the initial implementation for each instruction from the Z80 manual; the test suite (also AI-generated) caught edge cases – flag behaviour on overflow, the half-carry flag on DAA, interrupt timing. But mzx's most useful feature – built entirely through the AI feedback loop – is its headless CLI mode:

```
mzx --run program.bin@8000 --frames 100 --screenshot output.png
mzx --load code.bin@8000,data.bin@C000 --set PC=8000,SP=FFFF,EI,IM=1
mzx --model 128k --tap demo.tap --exec 'LOAD ""' --frames 500
mzx --run effect.bin@8000 --frames DI:HALT --dump-keyframes ./frames/
mzx --model pentagon --trd disk.trd --type "RUN\n" --screenshot grab.png
```

The `--run` flag loads a binary at a given address and starts execution – no ROM, no BASIC, no loading screen. The `--frames DI:HALT` trigger captures the screenshot at the exact moment the code signals "frame complete" by disabling interrupts before a HALT. The `--dump-keyframes` flag saves only frames where the screen changed – an automated visual regression test. The `--exec` and `--type` flags inject BASIC commands and keystrokes, allowing fully automated testing of programs that expect user interaction.

This book's screenshot pipeline uses mzx directly. Every code example screenshot in these pages was generated by:

```
sjasmplus --nologo --raw=build/example.bin example.a80
mzx --run build/example.bin@8000 --frames 50 --screenshot build/ch09_plasma.png
```

Twenty-one examples, zero manual intervention, reproducible with `make screenshots`.

The disassembler. `mzd` performs recursive descent analysis – the same technique used by IDA Pro. Given a binary, it traces all execution paths from entry points,

separates code from data, detects strings, generates cross-references, and auto-labels jump targets:

```
mzd illusion.bin --org $6000 --analyze --target spectrum --cycles --labels
```

The `--cycles` flag adds T-state counts to every instruction – automating the exact work that Introspec did by hand in his 2017 teardown of X-Trade's Illusion. The `--target spectrum` flag annotates system calls (RST $10 for character output, port $FE for border/keyboard). The `-R` flag produces reassemblable output, closing the disassemble-modify-reassemble loop.

The AI built both `mzd`'s instruction decoder (systematic table-driven work) and its analysis engine (recursive descent, control flow graph construction). The platform-specific ABI knowledge (which ZX Spectrum ROM calls do what) was partly AI-generated, partly pulled from existing documentation.

The standard library and peephole optimiser. Ten stdlib modules (maths, graphics, input, sound, etc.) and 35+ peephole patterns ("replace `LD A,0` with `XOR A`"). Both were AI-generated and human-refined. The AI knows the instruction set well enough to suggest valid simplifications; the human verifies semantic correctness.

Where AI Did Not Help Build MinZ

True Self-Modifying Code (TSMC). MinZ's most distinctive feature is TSMC – the compiler can emit code that rewrites its own instructions at runtime for performance. A single-byte opcode patch (7-20 T-states) replaces a conditional branch sequence (44+ T-states). The *concept* of TSMC was Alice's invention, not the AI's. The AI could not have proposed "what if the compiled code patched its own opcodes to change behaviour at runtime?" because the idea requires understanding both the compilation model and the Z80's instruction encoding at a level the AI does not reach unprompted.

The parser. MinZ originally used tree-sitter for parsing but hit out-of-memory issues on large files. The replacement – a hand-written recursive descent parser in Go – was designed by Alice, informed by AI consultation (GPT-4, o4-mini, and Claude were all asked for architectural advice). The AI colleagues agreed that a hand-written parser was the right approach and suggested keeping tree-sitter's test corpus. But the parser's actual grammar design – how MinZ's syntax maps to AST nodes – was human work. The AI could generate parser code for individual grammar rules but could not design the grammar itself.

Register allocation for the code generator. Deciding which variables live in which Z80 registers, when to spill to memory, and how to handle the Z80's irregular register file (only certain registers can be used with certain instructions) is a constraint satisfaction problem that the AI handles poorly. It generates code that works but wastes registers, uses unnecessary memory stores, and misses opportunities to keep hot values in registers across basic blocks.

The MinZ Verdict

MinZ could not exist without AI assistance. The sheer volume of systematic code – the instruction encoder, the emulator, the disassembler's analysis engine, the standard library, the peephole patterns – would have taken one developer years

to write manually. With AI assistance, MinZ went from concept to a four-tool ecosystem in roughly 18 months.

But MinZ's *interesting* features – TSMC, the zero-cost lambda-to-function transform, the UFCS method dispatch, mzx's `DI:HALT` trigger, mzd's platform-aware ABI annotations – are human inventions. The AI implemented them, but did not conceive them.

This maps precisely to the HiSoft C pattern. The tool accelerates the routine work enormously. The creative work remains human. The compromise is real and worth making.

23.6 Honest Take: "Z80 They Still Don't Know"

Introspec's scepticism about AI's Z80 capabilities is not generic technophobia. It comes from decades of experience pushing the Z80 to its absolute limits. When he says "Z80 they still don't know," he means something specific.

Consider the rotozoomer inner loop from his Illusion analysis. The effect walks through a texture at an angle, producing rotated and zoomed 2x2 chunky pixels. The inner loop is:

```
z80 id:ch23_honest_take_z80_they_still     ld   a, (hl)    ; 7T   read texture
byte     inc  l          ; 4T   next column (no carry needed: 256-aligned!)
dec  h          ; 4T   previous row     add  a,a        ; 4T   double (same as
SLA A but 4T not 8T)     add  a,a        ; 4T   quadruple     add  a,(hl)     ;
7T   combine with second texture sample                     ; --- 30T per pixel
pair
```

The key insight is `inc l` instead of `inc hl`. This saves 2 T-states but only works because the texture is aligned to a 256-byte boundary, so incrementing just L never needs to carry into H. The AI would use `inc hl` – the safe, general choice – and lose 2 T-states per iteration. Over 3,072 pixel pairs per frame, that is 6,144 T-states – nearly 10% of the frame budget on a 48K Spectrum.

This is what "Z80 they still don't know" means. Not that the AI cannot write Z80 assembly – it can. Not that it gets the instructions wrong – it usually does not. But that it does not understand the *context* in which each instruction choice matters. It does not think in budgets. It does not see that `inc l` is safe here because of the alignment constraint three abstraction layers up. It does not know that 2 T-states per iteration, multiplied by thousands of iterations, is the difference between fitting in one frame and needing two.

What the AI Gets Wrong, Specifically

We tested Claude Code on several Z80 tasks from the demoscene domain. Here are representative results.

Task: Write a DOWN_HL routine. DOWN_HL moves the screen pointer in HL one pixel row down in the Spectrum's interleaved video memory. It is the most commonly used graphics primitive on the platform. Introspec wrote a definitive article about it (Hype, 2020).

The AI's first attempt:

```
z80 id:ch23_what_the_ai_gets_wrong down_hl:     inc  h              ; 4T   move
down one pixel row     ld   a, h           ; 4T     and  7              ; 7T
ret  nz             ; 11T/5T  still within the character row     ld   a, l
; 4T     add  a, 32          ; 7T   next character row     ld   l, a           ;
4T     ret  c              ; 11T/5T  if carry, crossed into next third     ld
a, h           ; 4T     sub  8              ; 7T     ld   h, a           ; 4T
ret                 ; 10T
```

This is actually correct – it handles all three screen thirds properly, including the boundary transitions. The AI can get standard routines like DOWN_HL right because the pattern is well-documented and appears in many Z80 references. The routine works: `inc h` advances the pixel row, `add a, 32` advances the character row when needed, the carry from the L addition correctly detects third boundaries, and `sub 8` undoes the spurious TT increment for the common case.

But "correct" is not the same as "good." Introspec's article presents a version by RST7 using a dual-counter approach that handles all boundaries in 2,343 T-states for a full-screen traverse. The naive approach above – the standard textbook version – costs 5,922 T-states. The gap between "works" and "works well" is a factor of 2.5x, and the AI does not bridge that gap. It produces the first version any competent programmer would write, not the version an expert would optimise toward.

Task: Generate an unrolled screen fill. Asked to generate an unrolled PUSH-based screen fill (the technique from Chapter 3), the AI produced correct code – PUSH pairs writing two bytes at a time, DI/EI to protect the stack pointer manipulation. But it did not think to arrange the data in reverse order (PUSH writes high byte first, to lower addresses), which means the fill pattern was backwards. A human who has written PUSH fills before accounts for this automatically.

Task: Optimise a given inner loop. Given a working inner loop and asked to make it faster, the AI suggested standard optimisations: unrolling, lookup tables, register substitution. These are valid. But it did not find the non-obvious optimisation – the one where you rearrange memory layout to enable `inc l` instead of `inc hl`, or use the carry flag from an addition as a branch condition instead of a separate comparison. The non-obvious optimisation requires understanding the full context of the routine, and the AI's context window, while large, does not capture the *spatial* and *temporal* structure of a Z80 program the way a human expert's mental model does.

Where Introspec Is Right

The deepest Z80 optimisations are not about knowing instructions. They are about understanding the interplay between memory layout, register allocation, instruction encoding, timing constraints, and visual output – simultaneously. This interplay is what Introspec means by "evolving a computation scheme" (Chapter 1). A computation scheme is a holistic design where every decision affects every other decision. The AI operates on code locally. The expert operates on the scheme globally.

The AI does not know Z80 in the sense that Introspec knows Z80. It has memorised the instruction set but not internalised the machine.

Where Introspec Is Not Quite Right

But "Z80 they still don't know" implies the AI is useless for Z80 work, and that is not true either.

The AI is not trying to replace Introspec. It is trying to help Alice – a programmer who understands Z80 well enough to evaluate AI output but does not have decades of inner-loop optimisation experience. For Alice, the AI's output is a starting point that is better than a blank screen. She does not need the AI to find the `inc l` trick. She needs it to generate the first 80% of the routine so she can spend her time on the last 20%.

The demoscene has always been about the last 20%. The AI does not change that. It changes how fast you get through the first 80%.

23.7 The "Antique Toy" Demo: AI in Practice

The companion demo for this book – "Antique Toy" – is a deliberate experiment: build a ZX Spectrum demo with AI assistance and document what happens.

The name is a nod to Introspec's *Eager* (2015, 1st place at 3BM openair). We are implementing effects inspired by Eager – the attribute tunnel with 4-fold symmetry, the chaos zoomer, 4-phase colour animation – plus Dark's midpoint 3D engine from *Spectrum Expert* #02.

What has worked: Effect prototyping – Claude Code generates working first drafts fast enough to try ideas that would otherwise not be worth the typing time. "What if the tunnel used 8-fold symmetry instead of 4-fold?" takes 15 minutes with AI-generated code instead of 2 hours manually. Tooling – the build system, asset pipeline, Makefile rules, and test harnesses were all AI-generated and work reliably. Code review – feeding the AI a routine and asking "what is wrong?" catches obvious mistakes (off-by-one errors, forgotten DI/EI, wrong port numbers) before they cost debugging hours.

What has not worked: Dark's midpoint 3D engine. The virtual processor with packed 2-bit opcodes and 6-bit point numbers was incorrectly decoded. The averaging instruction computed `(A+B)/2` using `ADD A,B : SRA A`, which overflows for signed coordinates. Three debugging sessions, longer than writing it from scratch. Music integration failed similarly – the AI generated a player that conflicted with the effect code's use of shadow registers (EXX, EX AF,AF'). Both the player and the effect used shadow BC for different purposes, and the EXX in the interrupt handler swapped in stale values. This class of bug – system-level register conflicts across interrupt boundaries – requires understanding full system architecture, not just individual routines.

The honest assessment: "Antique Toy" is not finished. The effects work individually. Integration is ongoing. But AI assistance made the project *feasible* for a solo developer working evenings and weekends. The right question is not "does AI match a dedicated human team?" but "does AI assistance let more people make demos?" The answer, provisionally, is yes.

23.8 The Feedback Loop in Practice

A concrete example from the "Antique Toy" project: implementing 4-fold symmetry for the tunnel effect by copying the top-left 16x12 attribute quadrant to the other three quadrants with mirroring.

The prompt was specific: "Write a Z80 routine that copies the top-left 16x12 quadrant of the ZX Spectrum attribute area ($5800) to the other three quadrants with appropriate mirroring." Claude Code generated 47 lines that assembled on the first attempt.

Testing revealed the top-right quadrant was offset by one column. DeZog showed the problem: after the mirror loop decremented DE 16 times, the row-advance calculation forgot that DE had already been moved backwards. The code advanced DE by 32 (one row width) instead of the needed 48 (32 for the row + 16 to compensate for the mirror traversal). Pasting the register values into Claude Code – "After row 1, DE = $581F (should be $582F)" – produced the fix immediately. The bottom-right quadrant had the same error compounded. One more iteration fixed it.

Total: three iterations, roughly 25 minutes. Manual estimate for an experienced Z80 programmer: 40-60 minutes. For a newcomer: 2-3 hours. The AI saved time on initial generation. The debugging took the same time regardless of who wrote the code.

23.9 Building Your Own AI-Assisted Workflow

The practical setup: VS Code with Z80 Macro Assembler extension and Z80 Assembly Meter. Claude Code (or any code-capable LLM). An assembler (`mza` or sjasmplus). DeZog connected to an emulator. A Makefile.

The workflow: **Start with the AI** – describe what you want with specifics (target machine, memory addresses, assembler syntax). **Assemble immediately** – do not read the AI's code carefully; assemble it, paste errors back. **Test with border colours** – wrap AI-generated routines in the timing harness from Chapter 1. **Debug with DeZog** – set breakpoints, find the first register divergence, report it to the AI. **Iterate** – usually 2-5 rounds for moderate complexity; more than 5 means the AI is failing and you should write it yourself. **Optimise yourself** – once correct, profile and apply the techniques from Chapters 1-14.

Prompt Engineering for Z80

Good prompt: "Write a Z80 routine for ZX Spectrum 128K (Pentagon timing) that copies 16 bytes from the address in HL to screen memory at (DE), with the screen address following the Spectrum interleave pattern. After each byte, advance DE to the next pixel row using the standard down_hl method. Use mza syntax. Include cycle counts."

Bad prompt: "Write a sprite routine for the Spectrum."

The good prompt specifies machine, assembler, addresses, behaviour, and output format. The bad prompt leaves everything ambiguous, and the AI will fill the gaps

with wrong assumptions.

For optimisation prompts, give a concrete target: "This routine takes ~3,200 T-states. I need it under 2,400. Do not change the interface (HL = source, DE = destination, B = height). Pentagon timing." A performance target and interface constraint force the AI to look for real optimisations instead of restructuring the calling convention.

23.10 The Broader Picture

AI assistance does not change the abstraction level of the output – the Z80 still executes the same instructions at the same speeds. What it changes is the speed of the input: how fast you go from idea to working (if unoptimised) code. The demoscene's experts will still write better inner loops than any AI, but AI-assisted tooling lowers the entry barrier enough that more people can start making demos and learn the deep tricks for themselves.

Summary

- **AI-assisted Z80 development follows a feedback loop:** prompt, generate, assemble, test, debug, iterate. The AI generates the first draft fast; the human evaluates and refines. The loop typically takes 2-5 iterations for a routine of moderate complexity.
- **AI is reliable for** instruction encoding, cycle counts, boilerplate, dialect translation, and code explanation. It is moderately reliable for standard algorithms and simple self-modifying code. It is unreliable for novel optimisation, contended memory timing, creative effect design, and deep flag-based tricks.
- **DeZog integration** closes the gap between AI output and correct code. The human reads register states from the debugger and feeds divergences back to the AI, which reasons about the mismatch. Programmatic AI-debugger integration does not yet exist but is the obvious next step.
- **The MinZ case study** shows the pattern clearly: AI assistance made it possible for one developer to build a complete language toolchain (compiler, assembler, emulator, standard library) in 18 months. The routine work – instruction encoding, test generation, standard library functions – was AI-generated. The creative work – TSMC, zero-cost abstractions, grammar design – was human.
- **Introspec's scepticism is valid:** AI does not understand Z80 the way an expert does. It does not think in budgets, does not see cross-cutting constraints, does not find non-obvious optimisations. The deepest demoscene work remains beyond AI's reach.
- **The historical parallel holds:** HiSoft C was "10-15x faster than BASIC" but had no floats. AI-assisted Z80 development is dramatically faster for scaffolding and iteration but cannot match human experts for inner loop optimisation. Higher-level tools on constrained hardware have always been a compromise. The question is not "good or bad?" but "good *for what*?"

- **The practical workflow** combines Claude Code for code generation, DeZog for debugging, `mza` or sjasmplus for assembly, and a Makefile for build automation. Start with AI, assemble immediately, test with border colours, debug with DeZog, optimise yourself.
- **The broader effect** is positive: AI assistance lowers the entry barrier to Z80 development without lowering the ceiling. More people can start; the experts are still needed for the deep work. This is good for the demoscene.

Try It Yourself

1. **The boilerplate test.** Ask your AI assistant to generate a ZX Spectrum 128K boot template: ORG at $8000, disable interrupts, set up IM1, HALT loop with border colour timing harness. Assemble and run it. How many iterations did it take?
2. **The optimisation test.** Write (or AI-generate) a working attribute fill loop. Measure its cost with border colour timing. Then ask the AI to make it faster. Measure again. Now optimise it yourself using techniques from Chapters 1-3. Compare all three versions: original, AI-optimised, human-optimised.
3. **The DOWN_HL challenge.** Ask the AI to write a DOWN_HL routine. Test it on all 192 pixel rows. Does it handle the third-boundary transitions correctly? Compare to Introspec's analysis (Hype, 2020). This is a litmus test for AI Z80 competence.
4. **The MinZ experiment.** Install the MinZ toolchain (`mza`, `mzx`, `mzd`). Assemble a screen fill with `mza`, run it headless with `mzx --run fill.bin@8000 --frames 5 --screenshot fill.png`, then disassemble a demo binary with `mzd demo.bin --analyze --cycles --target spectrum`. Compare the AI-built disassembler's T-state counts to your own hand-counted totals from Chapter 1.
5. **The automated pipeline.** Write an effect, assemble it, and add it to a `Makefile` that runs `mzx --screenshot` for every binary. Run `mzx --dump-keyframes` to see exactly which frames produce visible changes. This is the same pipeline that generated every screenshot in this book.
6. **Build something.** Pick an effect from an earlier chapter. Use AI assistance to write the first draft. Iterate until it works. Profile it. Optimise the inner loop by hand. Document each step. You have just experienced the workflow this entire chapter describes.

This is the final technical chapter. What follows are the appendices – reference tables, setup guides, and the instruction reference you will reach for every time you write Z80 assembly.

> **Sources:** HiSoft C review (Spectrum Expert #02, 1998); Introspec "Technical Analysis of Illusion" (Hype, 2017); Introspec "DOWN_HL" (Hype, 2020); Introspec "GO WEST Parts 1-2" (Hype, 2015)

Glossary

Technical vocabulary used throughout *Coding the Impossible*. Terms are grouped by category; “First” indicates the chapter where the term is introduced or defined, “Also” lists chapters with significant usage.

A. Timing & Performance

Term	Definition	Canonical form	First	Also
T-state	One clock cycle of the Z80 CPU at 3.5 MHz (~286 ns). The fundamental unit of execution time on the Spectrum.	“T-states” in prose; “T” in code comments (e.g., `; 11T`)	Ch01	Ch02–Ch23
Frame	One complete display refresh at ~50 Hz (PAL). Duration varies by machine model.	“frame”	Ch01	all

Term	Definition	Canonical form	First	Also
Frame budget	Total T-states between interrupts: Pentagon 71,680, ZX 128K 70,908, ZX 48K 69,888. Practical budget after HALT + ISR + music player: ~66,000–68,000 (Pentagon), ~55,000–60,000 (128K with screen writes during active display). See Ch15 tact-maps for per-region breakdown (top border, active display, bottom border).	“frame budget”	Ch01	Ch04, Ch05, Ch08, Ch10, Ch12, Ch14, Ch16, Ch17, Ch18, Ch21
Scanline	One horizontal line of the display. Width varies: 224 T-states on 48K/Pentagon, 228 T-states on 128K. The frame consists of 320 (Pentagon), 311 (128K), or 312 (48K) scanlines including borders.	“scanline” (one word)	Ch01	Ch02, Ch08, Ch15, Ch17

Term	Definition	Canonical form	First	Also
Contended memory	RAM pages ($4000–$7FFF) where ULA and CPU share the memory bus on Sinclair hardware. CPU accesses are delayed by 0–6 extra T-states per access in a repeating 8-T-state pattern. Pentagon clones have no contention.	“contended memory”	Ch01	Ch04, Ch05, Ch15, Ch17, Ch18, Ch21, Ch22, Ch23
Machine cycle (M-cycle)	A group of 3–6 T-states within an instruction. The first M-cycle (M1) is always the opcode fetch at 4 T-states.	“machine cycle” or “M-cycle”	Ch01	–
Border time	Scanlines outside the 192-line active display (top/bottom borders). No contention; ~14,000 T-states available on 128K.	“border time”	Ch01	Ch08, Ch15, Ch17
Timing harness	Debugging technique: set border colour to red before code, black after; stripe height on screen shows T-state cost.	“border-colour timing harness”	Ch01	Ch02, Ch03, Ch08, Ch18

B. Hardware — Sinclair & Clones

Term	Definition	Canonical form	First	Also
Z80	Zilog Z80A CPU at 3.5 MHz. 8-bit data bus, 16-bit address bus. The processor in all ZX Spectrum models.	“Z80”	Ch01	all
ULA	Uncommitted Logic Array. Custom chip generating the video signal and handling I/O (keyboard, tape, speaker). Shares the memory bus with the CPU on Sinclair hardware.	“ULA”	Ch01	Ch02, Ch08, Ch15
ZX Spectrum 48K	Original Sinclair model. 69,888 T-states/frame, 312 scanlines, contended lower 32K.	“48K” or “ZX Spectrum 48K”	Ch01	Ch11, Ch15
ZX Spectrum 128K	Extended model with 128K banked RAM, AY sound chip, two display pages. 70,908 T-states/frame, 311 scanlines.	“128K” or “ZX Spectrum 128K”	Ch01	Ch11, Ch15, Ch20, Ch21
Screen memory	$4000–$5AFF. Pixel data ($4000–$57FF, 6,144 bytes) + attribute area ($5800–$5AFF, 768 bytes). Interleaved layout for pixel rows.	“screen memory”	Ch01	Ch02, Ch08, Ch16, Ch17

Term	Definition	Canonical form	First	Also
Attribute area	768 bytes at $5800–$5AFF. One byte per 8x8 character cell: FBPPPIII (Flash, Bright, Paper×3, Ink×3). Controls colour.	“attribute area”	Ch02	Ch08, Ch09, Ch10
Attribute clash	Hardware limitation: only 2 colours (ink + paper) per 8x8 cell. Overlapping sprites force colour compromises.	“attribute clash”	Ch02	Ch08, Ch16
Inter-leaved screen layout	Pixel rows in video memory are not sequential. Address encodes Y as 010TTSSS.LLLC-CCCC across two bytes. Three 2,048-byte thirds.	“interleaved screen layout”	Ch02	Ch07, Ch16, Ch17, Ch22
Shadow screen	Second display page at $C000 (bank 7) on 128K. Switched via port $7FFD bit 3.	“shadow screen”	Ch08	Ch10, Ch15, Ch21
AY-3-8910	General Instrument Programmable Sound Generator. Three square-wave tone channels, one noise generator, one envelope generator. 14 registers. Standard on 128K models.	“AY-3-8910” (first use), then “AY”	Ch11	Ch12, Ch15, Ch21
Port $FFFD / $BFFD	Ch11	Ch12		
“FFFD” / “$BFFD”	$BFFD|AY register select/data write ports on 128K.|			

Term	Definition	Canonical form	First	Also
Port $7FFD	128K memory paging and screen select port.	“$7FFD”	Ch08	Ch12, Ch15, Ch21
Port $FE|I/O port: bits 0–2 = border colour, bit 3 = MIC, bit 4 = EAR/speaker. Also keyboard input (active-low half-rows).|"FE”	Ch01	Ch02, Ch11, Ch18		
IM1 / IM2	Interrupt modes. IM1: handler at $0038. IM2: vectored via I register + data bus byte; used for custom interrupt handlers (music players, threading).	“IM1” / “IM2”	Ch01	Ch03, Ch05, Ch11, Ch12
DivMMC	SD card interface for modern Spectrum use. Supports esxDOS.	“DivMMC”	Ch15	Ch20
ZX Spectrum Next	FPGA-based enhanced Spectrum. Z80N CPU (extra instructions), Triple AY (3×AY), hardware sprites, copper co-processor, tilemap, 28 MHz turbo.	“ZX Spectrum Next” or “Next”	Ch11	Ch15

C. Hardware — Soviet/Post-Soviet Ecosystem

Term	Definition	Canonical form	First	Also
Pentagon 128	Most popular Soviet ZX Spectrum clone. No contended memory, 71,680 T-states/frame, 320 scanlines. The reference platform for the ZX Spectrum demoscene.	“Pentagon 128” (first use), then “Pentagon”	Ch01	Ch04, Ch05, Ch08, Ch11, Ch15, Ch16, Ch17, Ch18, Ch19, Ch20, Ch21, Ch22
Scorpion ZS-256	Soviet clone with TurboSound and extended memory. Pentagon-compatible timing.	“Scorpion ZS-256” (first use), then “Scorpion”	Ch04	Ch11, Ch15, Ch20
Tur-boSound	Two AY chips (2×AY) in one machine, providing 6 sound channels. Standard on Scorpion; available as add-on for Pentagon.	“TurboSound”	Ch11	Ch15, Ch20
TR-DOS	Disk operating system on Soviet clones via Beta Disk 128 interface. File format: `.trd`. The standard distribution format for demoscene compos and disk magazines.	“TR-DOS”	Ch15	Ch20
Beta Disk 128	Floppy controller interface standard on Pentagon and Scorpion clones.	“Beta Disk 128”	Ch15	Ch20

D. Hardware — Agon Light 2

Term	Definition	Canonical form	First	Also
Agon Light 2	Single-board computer with Zilog eZ80 CPU at 18.432 MHz and ESP32-based VDP. 512KB flat RAM, no banking.	“Agon Light 2” or “Agon”	Ch01	Ch11, Ch15, Ch18, Ch22
eZ80	Zilog eZ80 CPU. Z80-compatible instruction set; most instructions execute in fewer cycles. 24-bit flat addressing (ADL mode). ~368,640 T-states/frame at 50 Hz.	“eZ80”	Ch01	Ch15, Ch22
VDP	Video Display Processor. ESP32 microcontroller running FabGL library. Handles display, sprites, tilemaps, sound synthesis. Communicates with eZ80 over serial at 1,152,000 baud.	“VDP”	Ch02	Ch11, Ch15, Ch18, Ch22
ADL mode	24-bit addressing mode on the eZ80. Flat 512KB address space, no banking.	“ADL mode”	Ch15	Ch22
MOS	Operating system on the Agon. Provides API calls for VDP commands, keyboard, filesystem. `waitvblank` replaces Spectrum’s HALT for frame sync.	“MOS”	Ch18	Ch22

E. Techniques

Term	Definition	Canonical form	First	Also
Self-modifying code (SMC)	Writing to instruction bytes at runtime. Safe on Z80 (no instruction cache). Common patterns: patching immediate operands, changing jump targets, swapping opcodes.	“self-modifying code (SMC)” on first use; “SMC” subsequently	Ch03	Ch06, Ch07, Ch09, Ch10, Ch13, Ch16, Ch17, Ch22, Ch23
Unrolled loop	Trade code size for speed by repeating the loop body N times, eliminating loop counter overhead. Partial unrolling keeps DJNZ for the outer count.	“unrolled loop”	Ch02	Ch03, Ch07, Ch10, Ch16, Ch17
PUSH trick	Hijack SP to use PUSH for fast memory writes (5.5 T-states/byte vs 21 for LDIR). Must DI first to protect against interrupts using the stack.	“PUSH trick” or “stack-based output”	Ch03	Ch07, Ch08, Ch10, Ch16
LDI chain	Sequence of individual LDI instructions; 24% faster than LDIR for known-size copies. Combined with entry-point arithmetic for variable-length copies.	“LDI chain”	Ch03	Ch07, Ch09

Term	Definition	Canonical form	First	Also
LDPUSH	Fusing display data into `LD DE,nn : PUSH DE` executable code (21T per 2 bytes). Used in multicolor engines.	“LDPUSH”	Ch08	–
DOWN_HL	Classic routine to advance HL one pixel row down in the Spectrum’s interleaved screen memory. 20T common case, 77T worst case (third boundary).	“DOWN_HL”	Ch02	Ch16, Ch17, Ch23
RET-chaining	Set SP to a table of addresses; each routine ends with RET, dispatching to the next. 10T per dispatch vs 17T for CALL.	“RET-chaining”	Ch03	–
Code generation	Writing machine code into a buffer at runtime, then executing it. Eliminates branches and loop counters from inner loops.	“code generation”	Ch03	Ch06, Ch07, Ch09
Compiled sprites	Sprites compiled into a sequence of PUSH/LD instructions with pre-loaded register pairs. Fixed image, maximum speed.	“compiled sprites”	Ch03	Ch16
Double buffering	Maintaining two display pages to avoid tearing. On 128K: Screen 0 $(4000) and Screen 1 (C000)$, switched via port $7FFD.	“double buffering”	Ch05	Ch08, Ch10, Ch12

Term	Definition	Canonical form	First	Also
Dirty rectangles	Save/restore background under sprites before/after drawing. Avoid full-screen clears.	“dirty rectangles”	Ch08	Ch16, Ch21
Multicolor	Changing attribute values between ULA scanline reads to display more than 2 colours per 8x8 cell. Consumes 80–90% of CPU.	“multicolor”	Ch08	–
Page-aligned table	256-byte lookup table placed at an $xx00 address so H holds the base and L is the index. Single-register indexing with zero overhead.	“page-aligned table”	Ch04	Ch06, Ch07, Ch09, Ch10
Lookup table	Pre-computed table of values for fast runtime access. Avoids expensive calculations in inner loops.	“lookup table”	Ch03	Ch04, Ch07, Ch09, Ch17, Ch19, Ch20, Ch22
Split counters	Restructure screen iteration to match the three-level hierarchy (third/char row/scanline), eliminating branching. 60% faster than naive DOWN_HL traversal.	“split counters”	Ch02	–

Term	Definition	Canonical form	First	Also
4-phase colour	4-frame cycle (2 normal + 2 inverted attributes) at 50 Hz. Persistence of vision averages the colours, creating additional perceived colours per cell.	“4-phase colour”	Ch10	–
Digital drums	Digital PCM sample played through AY volume register as 4-bit DAC (attack phase), transitioning to AY envelope (decay phase). Consumes ~2 frames of CPU per hit.	“digital drums” or “hybrid drums”	Ch12	–
Asynchronous frame generation	Decoupling visual frame production from display via a ring buffer. Generator writes frames ahead; display reads at steady 50 Hz. Absorbs CPU bursts from drum playback.	“asynchronous frame generation”	Ch12	–

F. Assembly Notation & Directives

Term	Definition	Canonical form	First	Also
ORG	Assembler directive setting the code origin address. `ORG $8000` for Spectrum examples.	`ORG`	Ch01	all code examples

Term	Definition	Canonical form	First	Also
EQU	Define a named constant. `SCREEN EQU $4000`.	`EQU`	Ch02	all code examples
DB / DW / DS	Define Byte / Define Word / Define Space. `DB -3` (negative values allowed in sjasmplus).	`DB`, `DW`, `DS`	Ch04	all code examples
ALIGN	Align to a power-of-two boundary. sjasmplus directive.	`ALIGN`	Ch07	–
INCLUDE / INCBIN	Include source file / include binary data. sjasmplus directives.	`INCLUDE` / `INCBIN`	Ch14	Ch20, Ch21
DEVICE / SLOT / PAGE	sjasmplus directives for 128K memory banking emulation at assembly time.	`DEVICE`, `SLOT`, `PAGE`	Ch15	Ch20, Ch21
DISPLAY	sjasmplus directive printing a value at assembly time. Build-time diagnostics.	`DISPLAY`	Ch20	Ch21
`$FF`	Hex notation. `$` prefix is preferred. `#FF` also accepted by sjasmplus.	`$FF`	Ch01	all
`%10101010`	Binary notation.	`%10101010`	Ch01	all
`.label`	Local label, scoped to nearest enclosing global label.	`.label`	Ch01	all

Term	Definition	Canonical form	First	Also
sjasmplus	Primary assembler (v1.21.1). Full Z80/Z80N instruction set, macros, Lua scripting, DEVICE/SLOT/PAGE for banking, INCBIN, multiple output formats.	“sjasmplus”	Ch01	Ch14, Ch20, Ch21, Ch23

G. Demoscene & Culture

Term	Definition	Canonical form	First	Also
Compo	Competition at a demoparty. Categories include demo, intro (size-limited), music, graphics.	“compo”	Ch13	Ch20
Demoparty	Event where demoscene productions are shown and judged. Major ZX parties: Chaos Constructions, DiHalt, CAFe, Multimatograf (Russia); Revision, Forever (Western Europe).	“demoparty” or “party”	Ch20	–
NFO / file_id.diz	Text files bundled with demo releases containing credits, requirements, and sometimes technical notes.	“NFO” / “file_id.diz”	Ch20	–

Term	Definition	Canonical form	First	Also
Making-of	Post-release article documenting the development process and technical decisions of a demo. Published on Hype or in disk magazines.	“making-of”	Ch12	Ch20
Part / effect	A visual section of a demo (tunnel, scroller, sphere, etc.). Multiple parts are sequenced by a demo engine.	“part” or “effect”	Ch09	Ch12, Ch20
Scripting engine	System that sequences demo parts and synchronises them to music. Two-level: outer script (effect sequence) + inner script (parameter variation within an effect).	“scripting engine”	Ch09	Ch12, Ch20
kWORK	Introspec’s command: “generate N frames, then show them independently of generation.” The bridge between scripting and asynchronous frame generation.	“kWORK”	Ch09	Ch12

Term	Definition	Canonical form	First	Also
Zapilator	Russian scene slang for a “precalculator” demo – one that pre-computes all frames before playback. Carries mild disapproval (implies no real-time rendering).	“zapilator”	Ch09	–

Key People

Name	Role	Chapters
Dark	Coder, X-Trade. Author of Spectrum Expert programming articles. Coder of Illusion.	Ch01, Ch04, Ch05, Ch06, Ch07, Ch10
Introspec (spke)	Coder, Life on Mars. Reverse-engineered Illusion. Authored Hype articles (technical analyses, GO WEST series, DOWN_HL). Coded Eager demo.	Ch01–Ch12, Ch15, Ch23
n1k-o	Musician, Skrju. Composed Eager soundtrack. Developed hybrid drum technique with Introspec.	Ch09, Ch12
diver4d	Coder, 4th Dimension. GABBA demo (CAFe 2019). Pioneered video-editor sync workflow (Luma Fusion).	Ch09, Ch12
DenisGrachev	Coder. Old Tower, GLUF engine, Ringo engine, Dice Legends. RET-chaining technique.	Ch03, Ch08

Name	Role	Chapters
Robus	Coder. Z80 threading system, WAYHACK demo.	Ch12
sq (psndcj)	Coder. Chunky pixel optimisation research (Born Dead #05, Hype). Development environment guide.	Ch01, Ch07
RST7	Coder. Dual-counter DOWN_HL optimisation.	Ch02

Key Demos & Productions

Name	Author/Group	Year	Chapters
Illusion	X-Trade (Dark)	1996	Ch01, Ch04, Ch05, Ch06, Ch07, Ch10
Eager (to live)	Introspec / Life on Mars	2015	Ch02, Ch03, Ch09, Ch10, Ch12
GABBA	diver4d / 4th Dimension	2019	Ch12
WAYHACK	Robus	–	Ch12
Old Tower	DenisGrachev	–	Ch08
Lo-Fi Motion	–	–	Ch20

Key Publications

Name	Description	Chapters
Spectrum Expert (#01–#02)	Russian ZX disk magazine (1997–98). Dark's programming tutorials.	Ch01, Ch04, Ch05, Ch06, Ch10, Ch11
Hype	Russian online demoscene platform (hype.retroscene.org). Technical articles, making-ofs, debates.	Ch01–Ch12, Ch23
Born Dead	Russian demoscene newspaper. sq's chunky pixel research (#05, ~2001).	Ch07
Black Crow	Russian ZX scene magazine. Early multicolor documentation (#05, ~2001).	Ch08

H. Algorithms & Compression

Term	Definition	Canonical form	First	Also
Shift-and-add multiply	Classic 8×8 unsigned multiply. Scan multiplier bits, shift-and-add. 196–204 T-states.	“shift-and-add multiply”	Ch04	Ch05
Square-table multiply	$A \times B = ((A+B)^2 - (A-B)^2)/4$ via lookup. ~61 T-states. Trades 512 bytes of tables for speed.	“square-table multiply”	Ch04	Ch05, Ch07
Logarithmic division	$\log(A/B) = \log(A) - \log(B)$. Two table lookups + subtraction. ~50–70 T-states. Low precision.	“logarithmic division”	Ch04	Ch05
Parabolic sine approximation	Approximate sine with parabola $y = 1 - 2(x/\pi)^2$. Max error ~5.6%. 256-byte table, signed.	“parabolic sine approximation”	Ch04	Ch05, Ch06, Ch09
Bresenham line drawing	Step along major axis with error accumulator for minor-axis steps. ~80 T-states/pixel naive; ~48 with Dark’s 8×8 matrix method.	“Bresenham”	Ch04	–
Midpoint method	Rotate only basis vertices fully; derive remaining vertices by averaging. ~36 T-states per derived vertex vs ~2,400 for full rotation.	“midpoint method”	Ch05	–
Fixed-point arithmetic	Represent fractional values in integer registers. Common format: 8.8 (8-bit integer + 8-bit fraction).	“fixed-point”	Ch04	Ch05, Ch18, Ch19

Term	Definition	Canonical form	First	Also
AABB collision	Axis-Aligned Bounding Box overlap test. Four comparisons: left-right and top-bottom for two rectangles. ~70–156 T-states.	“AABB”	Ch19	Ch21
Backface culling	Skip back-facing polygons using cross-product normal Z-component test. ~500 T-states per face.	“backface culling”	Ch05	–
ZX0	Compression format by Einar Saukas. Excellent ratio, moderate decompression speed.	“ZX0”	Ch14	Ch20
LZ4	Fast decompression (~34 T-states/byte). The choice for real-time streaming.	“LZ4”	Ch14	–
Exomizer	High-ratio compressor for 8-bit platforms. Slow decompression.	“Exomizer”	Ch14	–
MegaLZ	Compression format. Good ratio/speed balance.	“MegaLZ”	Ch14	–
hrust1	Compression format common in Russian demoscene.	“hrust1”	Ch14	Ch20

This glossary is extracted from all 23 chapters of “Coding the Impossible.” For detailed treatment of any term, see the chapter listed under “First.”

Appendix A: Z80 Instruction Quick Reference

> *"An instruction that should take 7 T-states might take 13 if it lands on the worst phase of the contention cycle."* – Chapter 1

This is not a complete Z80 manual. It is a reference card for demoscene and game programmers on ZX Spectrum and Agon Light 2 – the instructions you actually use, the timings you need to know by heart, and the patterns that save T-states in inner loops.

All T-state counts assume **Pentagon timing** (no contention). Byte counts are the instruction encoding length. Flag columns show: **S** (sign), **Z** (zero), **H** (half-carry), **P/V** (parity/overflow), **N** (subtract), **C** (carry). A dash means unchanged; a dot means undefined.

8-Bit Load Instructions

Instruction	Bytes	T-states	Flags	Notes
`LD r,r'`	1	4	——	Fastest instruction. r,r' = A,B,C,D,E,H,L
`LD r,n`	2	7	——	Immediate load
`LD r,(HL)`	1	7	——	Memory read via HL
`LD (HL),r`	1	7	——	Memory write via HL
`LD (HL),n`	2	10	——	Immediate to memory
`LD A,(BC)`	1	7	——	
`LD A,(DE)`	1	7	——	
`LD (BC),A`	1	7	——	
`LD (DE),A`	1	7	——	
`LD A,(nn)`	3	13	——	Absolute address

Instruction	Bytes	T-states	Flags	Notes
`LD (nn),A`	3	13	——	Absolute address
`LD r,(IX+d)`	3	19	——	Indexed. Expensive – avoid in inner loops
`LD (IX+d),r`	3	19	——	Indexed
`LD (IX+d),n`	4	23	——	Indexed immediate
`LD A,I`	2	9	SZ0P0-	P/V = IFF2
`LD A,R`	2	9	SZ0P0-	P/V = IFF2; R = refresh counter

16-Bit Load Instructions

Instruction	Bytes	T-states	Flags	Notes
`LD rr,nn`	3	10	——	rr = BC, DE, HL, SP
`LD HL,(nn)`	3	16	——	
`LD (nn),HL`	3	16	——	
`LD rr,(nn)`	4	20	——	rr = BC, DE, SP (ED prefix)
`LD (nn),rr`	4	20	——	rr = BC, DE, SP (ED prefix)
`LD SP,HL`	1	6	——	Set up stack pointer
`LD SP,IX`	2	10	——	
`PUSH rr`	1	11	——	rr = AF, BC, DE, HL. **5.5T per byte**
`POP rr`	1	10	——	**5T per byte** – fastest 2-byte read
`PUSH IX`	2	15	——	
`POP IX`	2	14	——	

8-Bit Arithmetic and Logic

Instruction	Bytes	T-states	Flags	Notes
`ADD A,r`	1	4	SZ.V0C	
`ADD A,n`	2	7	SZ.V0C	
`ADD A,(HL)`	1	7	SZ.V0C	
`ADC A,r`	1	4	SZ.V0C	Add with carry
`ADC A,n`	2	7	SZ.V0C	
`SUB r`	1	4	SZ.V1C	
`SUB n`	2	7	SZ.V1C	
`SUB (HL)`	1	7	SZ.V1C	

Instruction	Bytes	T-states	Flags	Notes
`SBC A,r`	1	4	SZ.V1C	Subtract with carry
`CP r`	1	4	SZ.V1C	Compare (SUB without storing result)
`CP n`	2	7	SZ.V1C	
`CP (HL)`	1	7	SZ.V1C	
`AND r`	1	4	SZ1P00	H always set, C always cleared
`AND n`	2	7	SZ1P00	
`OR r`	1	4	SZ0P00	Clears H and C
`OR n`	2	7	SZ0P00	
`XOR r`	1	4	SZ0P00	`XOR A` = zero A in 4T/1B (vs `LD A,0` = 7T/2B)
`XOR n`	2	7	SZ0P00	
`INC r`	1	4	SZ.V0-	Does **not** affect carry
`DEC r`	1	4	SZ.V1-	Does **not** affect carry
`INC (HL)`	1	11	SZ.V0-	Read-modify-write
`DEC (HL)`	1	11	SZ.V1-	Read-modify-write
`NEG`	2	8	SZ.V1C	A = 0 - A (two’s complement negate)
`DAA`	1	4	SZ.P-C	BCD adjust – rarely used in demos
`CPL`	1	4	–1-1-	A = NOT A (one’s complement)
`SCF`	1	4	–0-00	Set carry flag. N,H cleared. New behaviour on CMOS

Instruction	Bytes	T-states	Flags	Notes
`CCF`	1	4	–.-0.	Complement carry. H = old C

16-Bit Arithmetic

Instruction	Bytes	T-states	Flags	Notes
`ADD HL,rr`	1	11	–.?0C	rr = BC, DE, HL, SP. Only affects H, N, C
`ADC HL,rr`	2	15	SZ.V0C	Full flag set
`SBC HL,rr`	2	15	SZ.V1C	Full flag set
`INC rr`	1	6	——	No flags affected
`DEC rr`	1	6	——	No flags affected
`ADD IX,rr`	2	15	–.?0C	rr = BC, DE, IX, SP

Key point: `INC rr` and `DEC rr` do **not** set the zero flag. You cannot use `DEC BC / JR NZ` as a 16-bit loop counter. Use `DEC B / JR NZ` for 8-bit loops with `DJNZ`, or test BC explicitly.

Rotate and Shift

Instruction	Bytes	T-states	Flags	Notes
`RLCA`	1	4	–0-0C	Rotate A left, bit 7 to carry and bit 0
`RRCA`	1	4	–0-0C	Rotate A right, bit 0 to carry and bit 7
`RLA`	1	4	–0-0C	Rotate A left through carry

Instruction	Bytes	T-states	Flags	Notes
`RRA`	1	4	–0-0C	Rotate A right through carry. **Key for multiply loops**
`RLC r`	2	8	SZ0P0C	CB-prefix. Full flag set
`RRC r`	2	8	SZ0P0C	
`RL r`	2	8	SZ0P0C	Rotate left through carry
`RR r`	2	8	SZ0P0C	Rotate right through carry
`SLA r`	2	8	SZ0P0C	Shift left arithmetic. Bit 0 = 0
`SRA r`	2	8	SZ0P0C	Shift right arithmetic. Bit 7 preserved (sign extend)
`SRL r`	2	8	SZ0P0C	Shift right logical. Bit 7 = 0
`RLC (HL)`	2	15	SZ0P0C	Read-modify-write
`RL (HL)`	2	15	SZ0P0C	Scroll pixel data left
`RR (HL)`	2	15	SZ0P0C	Scroll pixel data right
`SLA (HL)`	2	15	SZ0P0C	
`SRL (HL)`	2	15	SZ0P0C	
`RLD`	2	18	SZ0P0-	Rotate (HL) nibbles left through A. Useful for nybble graphics
`RRD`	2	18	SZ0P0-	Rotate (HL) nibbles right through A

Demoscene note: `RLA`/`RRA` (4T, 1 byte) only affect carry and bits 3,5 of F. The CB-prefix versions `RL r`/`RR r` (8T, 2 bytes) set all flags. In multiply loops, the accumulator versions save half the cost.

Bit Manipulation

Instruction	Bytes	T-states	Flags	Notes
`BIT b,r`	2	8	.Z1.0-	Test bit b of register
`BIT b,(HL)`	2	12	.Z1.0-	Test bit b of memory
`SET b,r`	2	8	——	Set bit b of register
`SET b,(HL)`	2	15	——	Set bit b of memory. **Used in line drawing**
`RES b,r`	2	8	——	Reset bit b of register
`RES b,(HL)`	2	15	——	Reset bit b of memory

Jump, Call, Return

Instruction	Bytes	T-states	Flags	Notes
`JP nn`	3	10	——	Absolute jump
`JP cc,nn`	3	10	——	Conditional: NZ, Z, NC, C, PO, PE, P, M. **Same speed taken or not**
`JR e`	2	12	——	Relative jump (-128 to +127)
`JR cc,e`	2	12/7	——	cc = NZ, Z, NC, C only. **7T if not taken**

Instruction	Bytes	T-states	Flags	Notes
`JP (HL)`	1	4	——	Jump to address in HL. Fastest indirect jump
`JP (IX)`	2	8	——	Jump to address in IX
`DJNZ e`	2	13/8	——	Dec B, jump if NZ. **13T taken, 8T not taken**
`CALL nn`	3	17	——	Push PC, jump to nn
`CALL cc,nn`	3	17/10	——	10T if not taken
`RET`	1	10	——	Pop PC. **Used for RET-chaining dispatch**
`RET cc`	1	11/5	——	5T if not taken
`RST p`	1	11	——	Call to $00,$08,$10,$18,$20,$28

Key comparisons for dispatch:

Method	T-states	Bytes
`CALL nn`	17	3
`RET` (as dispatch in RET-chain)	10	1
`JP (HL)`	4	1
`JP nn`	10	3

I/O Instructions

Instruction	Bytes	T-states	Flags	Notes
`OUT (n),A`	2	11	——	Port address = `(A << 8) | n`. Border: `OUT ($FE),A`

Instruction	Bytes	T-states	Flags	Notes
`IN A,(n)`	2	11	——	Port address = `(A << 8) | n`. Keyboard: `IN A,($FE)`
`OUT (C),r`	2	12	——	Port address = BC. **AY register write**
`IN r,(C)`	2	12	SZ0P0-	Port address = BC. Sets flags
`OUTI`	2	16	.Z..1.	Out (HL) to port (C), inc HL, dec B
`OTIR`	2	21/16	01..1.	Repeat OUTI until B=0. 16T on last
`OUTD`	2	16	.Z..1.	Out (HL) to port (C), inc HL, dec B

AY-3-8910 port addresses on ZX Spectrum 128K:

Port	Address	Purpose
Register select	`$FFFD`	`LD BC,$FFFD : OUT (C),A`
Data write	`$BFFD`	`LD B,$BF : OUT (C),r`
Data read	`$FFFD`	`IN A,(C)`

Typical AY register write sequence (24T + overhead):

```
ld   bc, $FFFD      ; 10T  AY register select port
out  (c), a          ; 12T  select register number (in A)
ld   b, $BF          ;  7T  switch to data port $BFFD
out  (c), e          ; 12T  write value (in E)
                     ; --- 41T total
```

Block Instructions

Instruction	Bytes	T-states	Flags	Notes
`LDI`	2	16	–0.0-	(DE) = (HL), inc HL, inc DE, dec BC. P/V = (BC != 0)
`LDIR`	2	21/16	–000-	Repeat LDI. 21T per byte, 16T last byte
`LDD`	2	16	–0.0-	(DE) = (HL), dec HL, dec DE, dec BC
`LDDR`	2	21/16	–000-	Repeat LDD. 21T per byte, 16T last byte
`CPI`	2	16	SZ.?1-	Compare A with (HL), inc HL, dec BC
`CPIR`	2	21/16	SZ.?1-	Repeat CPI. Stops when match or BC=0
`CPD`	2	16	SZ.?1-	Compare A with (HL), dec HL, dec BC
`CPDR`	2	21/16	SZ.?1-	Repeat CPD

LDI vs LDIR per-byte cost:

Method	Per byte	256 bytes	32 bytes	Savings
LDIR	21T (16T last)	5,371T	672T	–
LDI chain	16T	4,096T	512T	24% faster

An unrolled LDI chain costs 2 bytes per LDI (`$ED $A0`), but saves 5T per byte – 24% faster than LDIR. See Chapter 3 for entry-point arithmetic with LDI chains.

Exchange and Misc

Instruction	Bytes	T-states	Flags	Notes
`EX DE,HL`	1	4	——	Swap DE and HL. **Free pointer swap**
`EX AF,AF'`	1	4	——	Swap AF with shadow AF’
`EXX`	1	4	——	Swap BC,DE,HL with BC’,DE’,HL’. **6 registers in 4T**
`EX (SP),HL`	1	19	——	Swap HL with top of stack. Useful for parameter passing
`EX (SP),IX`	2	23	——	Swap IX with top of stack
`DI`	1	4	——	Disable interrupts. **Required before stack tricks**
`EI`	1	4	——	Enable interrupts. Delayed one instruction
`HALT`	1	4+	——	Wait for interrupt. The frame sync instruction
`NOP`	1	4	——	Padding, timing
`IM 1`	2	8	——	Interrupt mode 1 (RST $38). Standard Spectrum mode

Instruction	Bytes	T-states	Flags	Notes
`IM 2`	2	8	——	Interrupt mode 2. Uses I register as vector table high byte

---

The Demoscene "Fast" Instructions

These are the cheapest instructions per category – the building blocks of every optimised inner loop.

Fastest Register-to-Register Move

`LD r,r'` – **4T, 1 byte**. The minimum cost of any Z80 instruction. Includes `LD A,A` (effectively a NOP that does not affect flags).

Fastest Way to Zero a Register

`XOR A` – **4T, 1 byte**. Sets A to zero, sets Z flag, clears carry. Compare with `LD A,0` at 7T/2 bytes. Always use `XOR A` unless you need flags preserved.

Fastest Memory Read

`LD A,(HL)` – **7T, 1 byte**. The minimum cost for any memory read. Other register sources (`LD A,(BC)`, `LD A,(DE)`) are also 7T/1 byte, but HL is the only pointer that supports `LD r,(HL)` for all registers.

Fastest Memory Write

`LD (HL),r` – **7T, 1 byte**. Tied with read. Writing to a BC or DE pointer (`LD (BC),A`, `LD (DE),A`) is also 7T/1B but only works with A.

Fastest 2-Byte Write

`PUSH rr` – **11T, 1 byte** for 2 bytes = **5.5T per byte**. The fastest way to write data to memory, but only to where SP points (downward). Requires DI and hijacking the stack pointer. See Chapter 3.

Fastest 2-Byte Read

`POP rr` – **10T, 1 byte** for 2 bytes = **5T per byte**. Even faster than PUSH for reading. Use with SP pointing at a data table for ultra-fast lookups.

Fastest Block Copy

Method	Per byte	Notes
`PUSH/POP` pair	5.25T	Write: 5.5T, Read: 5T. But needs SP hijack
`LDI` (unrolled chain)	16T	No setup per byte. 24% faster than LDIR
`LDIR`	21T	Single instruction, but slow per byte
`LD (HL),r` + `INC HL`	13T	Manual loop body (no loop counter)
`LD (HL),r` + `INC L`	11T	Only works within 256-byte page

Fastest I/O

`OUT (n),A` – **11T, 2 bytes**. For fixed port addresses (border, etc.). For variable ports (AY), `OUT (C),r` at 12T/2 bytes is the only option.

Fastest Pointer Swap

`EX DE,HL` – **4T, 1 byte**. Exchange DE and HL contents instantly. No other register swap is this cheap. `EXX` also 4T/1 byte but swaps all three pairs at once.

Fastest Conditional Loop

`DJNZ e` – **13T taken, 8T not taken, 2 bytes**. Decrements B and jumps. Compare with `DEC B / JR NZ,e` at 4+12 = 16T/3 bytes. DJNZ saves 3T and 1 byte per iteration.

Fastest Indirect Jump

`JP (HL)` – **4T, 1 byte**. Jumps to the address in HL. Despite the misleading mnemonic, this does NOT read from memory at (HL) – it loads PC with the value of HL. Indispensable for jump tables and computed gotos.

Undocumented Instructions

These instructions are not in the official Zilog documentation but work reliably on all Z80 silicon (NMOS and CMOS), all ZX Spectrum clones, and the eZ80. They are widely used in demoscene code and supported by sjasmplus.

IXH, IXL, IYH, IYL (Half-Index Registers)

The IX and IY registers can be split into 8-bit halves, accessed by prefixing normal H/L instructions with DD/FD:

Instruction	Bytes	T-states	Notes
`LD A,IXH`	2	8	Read high byte of IX
`LD A,IXL`	2	8	Read low byte of IX
`LD IXH,A`	2	8	Write high byte of IX
`LD IXL,n`	3	11	Immediate into IX low
`ADD A,IXL`	2	8	Arithmetic with IX halves
`INC IXH`	2	8	Increment IX high byte
`DEC IXL`	2	8	Decrement IX low byte

Demoscene use: Two extra 8-bit registers for counters, accumulators, or small values without touching the main register file. Particularly useful when BC/DE/HL are all occupied as pointers. Cost: 4T more than the equivalent main-register operation.

sjasmplus syntax: `IXH`, `IXL`, `IYH`, `IYL` (also accepts `HX`, `LX`, `HY`, `LY`).

SLL r (Shift Left Logical)

Instruction	Bytes	T-states	Notes
`SLL r`	2	8	Shift left, bit 0 set to 1 (not 0)
`SLL (HL)`	2	15	Same for memory

`SLL` shifts left and sets bit 0 to 1 (unlike `SLA` which sets bit 0 to 0). Opcode: CB 30+r. Occasionally useful for constructing bit patterns.

sjasmplus syntax: `SLL` or `SLI` or `SL1`.

OUT (C),0

Instruction	Bytes	T-states	Notes
`OUT (C),0`	2	12	Output zero to port BC

Opcode `ED 71`. Outputs zero to the port addressed by BC. On CMOS Z80s (including eZ80), this outputs $FF instead. **Not portable to Agon Light 2.** On NMOS Z80s (all real Spectrums), it outputs $00 reliably.

sjasmplus syntax: `OUT (C),0`.

CB-Prefix Undocumented Bit Operations on (IX+d)

Instructions like `SET b,(IX+d),r` simultaneously perform a bit operation on memory at (IX+d) and copy the result into register r. These are 4-byte instructions (DD CB dd op) taking 23T. Occasionally useful but rarely critical.

Flag Effects Cheat Sheet

Knowing which instructions set which flags lets you avoid redundant `CP` or `AND A` instructions – a common source of wasted T-states.

Instructions That Set All Arithmetic Flags (S, Z, H, P/V, N, C)

- `ADD A,r/n/(HL)` – P/V = overflow
- `ADC A,r/n/(HL)` – P/V = overflow
- `SUB r/n/(HL)` – P/V = overflow
- `SBC A,r/n/(HL)` – P/V = overflow
- `CP r/n/(HL)` – Same flags as SUB but A unchanged
- `NEG` – P/V = overflow
- `ADC HL,rr` – P/V = overflow
- `SBC HL,rr` – P/V = overflow

Instructions That Set Z and S (but NOT Carry)

- `INC r` / `DEC r` – C unchanged. **Cannot test carry after INC/DEC.**
- `INC (HL)` / `DEC (HL)` – Same
- `AND r/n/(HL)` – C always 0, H always 1
- `OR r/n/(HL)` – C always 0, H always 0
- `XOR r/n/(HL)` – C always 0, H always 0
- `IN r,(C)` – C unchanged
- `BIT b,r/(HL)` – Z = complement of tested bit, C unchanged
- All CB-prefix rotates/shifts – Full flag set including C

Instructions That Set ONLY Carry-Related Flags

- `ADD HL,rr` – H and C only (S, Z, P/V unchanged)
- `RLCA` / `RRCA` / `RLA` / `RRA` – C, H=0, N=0 only (S, Z, P/V unchanged)
- `SCF` – C=1, H=0, N=0
- `CCF` – C inverted, H = old C, N=0

Instructions That Set NO Flags

- `LD` (all variants)
- `INC rr` / `DEC rr` (16-bit inc/dec)
- `PUSH` / `POP` (except POP AF restores flags)
- `EX` / `EXX`
- `DI` / `EI` / `HALT` / `NOP`
- `JP` / `JR` / `DJNZ` / `CALL` / `RET` / `RST`
- `OUT (n),A` / `IN A,(n)` (the non-CB versions)

Practical Tricks

Test A for zero without CP 0:

```
or   a              ; 4T  sets Z if A=0, clears C
and  a              ; 4T  same effect, but also sets H
```

Test carry after 16-bit INC/DEC: You cannot. `INC rr`/`DEC rr` set no flags. To test if a 16-bit register reached zero:

```
ld   a, b          ; 4T
or   c             ; 4T  Z set if BC = 0
```

Skip CP after SUB: If you already performed `SUB r`, the flags are set – do not follow it with `CP` or `OR A`.

INC/DEC preserve carry: Use `INC r`/`DEC r` between multi-precision arithmetic without destroying the carry chain.

Register Architecture

Main Register Set

```
A   F          Accumulator + Flags
B   C          Counter (B for DJNZ) + general
D   E          General purpose pair
H   L          Primary memory pointer (HL is the "accumulator pair")
```

Special Registers

```
SP             Stack pointer (16-bit)
PC             Program counter (16-bit)
IX             Index register (16-bit, DD prefix, +4T penalty)
IY             Index register (16-bit, FD prefix, +4T penalty)
               NOTE: IY is used by the Spectrum ROM interrupt handler.
               Do not use IY unless you have DI or IM2 set up.
I              Interrupt vector page (used in IM2)
R              Refresh counter (7-bit, increments every M1 cycle)
```

Shadow Registers

```
A'  F'         Swapped with EX AF,AF'
B'  C'         \
D'  E'          | Swapped all three with EXX
H'  L'         /
```

`EXX` swaps BC/DE/HL with BC’/DE’/HL’ in **4T**. This gives you six extra 8-bit registers (or three extra 16-bit pairs) at virtually zero cost. Common use: keep pointers in the shadow set and swap in/out as needed.

Warning: The Spectrum ROM interrupt handler (IM1) uses the shadow registers. If interrupts are enabled, `EXX`/`EX AF,AF'` data will be corrupted on every interrupt. Always `DI` before using shadow registers, or switch to IM2 with your own handler.

Register Pairing for Instructions

Pair	Used by	Notes
BC	`DJNZ` (B only), `OUT (C),r`, `IN r,(C)`, block instructions (counter)	B = loop counter, C = port low byte

Pair	Used by	Notes
DE	`EX DE,HL`, `LDI`/`LDIR` (destination), `LD (DE),A`	Destination pointer for block ops
HL	Nearly everything: `LD r,(HL)`, `ADD HL,rr`, `JP (HL)`, `PUSH`/`POP`, `LDI` (source)	The universal pointer
AF	`PUSH AF`/`POP AF`, `EX AF,AF'`	A = accumulator, F = flags
SP	`PUSH`/`POP`, `LD SP,HL`, `EX (SP),HL`	Hijack for data tricks

Common Instruction Sequences

Pixel Address Calculation (Screen Address from Y,X)

Convert screen coordinates to ZX Spectrum video memory address. Input: B = Y (0-191), C = X (0-255). Output: HL = screen byte address, A = bit mask.

```
; pixel_addr: calculate screen address from coordinates
; Input:  B = Y (0-191), C = X (0-255)
; Output: HL = byte address, A = pixel bit position
; Cost:   ~55 T-states
;
pixel_addr:
    ld   a, b           ; 4T   A = Y
    and  $07             ; 7T   scanline within char cell (SSS)
    or   $40             ; 7T   add screen base ($4000 high byte)
    ld   h, a           ; 4T   H = 010 00 SSS (partial)
    ld   a, b           ; 4T   A = Y again
    rra                 ; 4T   \
    rra                 ; 4T    | shift character row (TTRR RRR)
    rra                 ; 4T   /  to bits 4-3
    and  $18             ; 7T   mask TT (third bits)
    or   h              ; 4T   H = 010 TT SSS
    ld   h, a           ; 4T
    ld   a, c           ; 4T   A = X
    rra                 ; 4T   \
    rra                 ; 4T    | X / 8
    rra                 ; 4T   /
    and  $1F             ; 7T   mask to 5-bit column
    ld   l, a           ; 4T   L = 000 CCCCC
```

DOWN_HL: Move One Pixel Row Down

The most-used graphics primitive on the Spectrum. Common case (within character cell) costs only 20T.

```
; down_hl: advance HL one pixel row down
; Input:  HL = screen address
; Output: HL = address one row below
```

```
; Cost:   20T (common), 46T (third boundary), 77T (char boundary)
;
down_hl:
    inc  h              ; 4T   try next scanline
    ld   a, h           ; 4T
    and  7              ; 7T   crossed character boundary?
    ret  nz             ; 5T   no: done (20T total)

    ld   a, l           ; 4T   yes: advance character row
    add  a, 32          ; 7T   L += 32
    ld   l, a           ; 4T
    ret  c              ; 5T   carry = crossed third (46T total)

    ld   a, h           ; 4T   same third: undo extra H increment
    sub  8              ; 7T
    ld   h, a           ; 4T
    ret                 ; 10T  (77T total)
```

8x8 Unsigned Multiply (Shift-and-Add)

From Dark / X-Trade, Spectrum Expert #01 (1997). Used in rotation matrices and coordinate transforms.

```
; mulu112: 8x8 unsigned multiply
; Input:  B = multiplicand, C = multiplier
; Output: A:C = 16-bit result (A = high, C = low)
; Cost:   196-204 T-states
;
mulu112:
    ld   a, 0           ; 7T   clear accumulator
    ld   d, 8           ; 7T   8 bits

.loop:
    rr   c              ; 8T   shift multiplier bit into carry
    jr   nc, .noadd     ; 7/12T
    add  a, b           ; 4T   add multiplicand
.noadd:
    rra                 ; 4T   shift result right
    dec  d              ; 4T
    jr   nz, .loop      ; 12T
    ret                 ; 10T
```

AY Register Write

Standard sequence for writing to the AY-3-8910 sound chip on ZX Spectrum 128K.

```
; ay_write: write value to AY register
; Input:  A = register number (0-15), E = value
; Cost:   41 T-states (plus CALL/RET overhead)
;
ay_write:
    ld   bc, $FFFD      ; 10T  register select port
    out  (c), a          ; 12T  select register
```

```
    ld   b, $BF          ;  7T  data port ($BFFD)
    out  (c), e          ; 12T  write value
    ret                  ; 10T
```

16-Bit Compare (HL vs DE)

The Z80 has no direct 16-bit compare. Use `SBC` and restore.

```
; Compare HL with DE (sets flags as if HL - DE)
; Destroys: A (if using the OR method for equality)
;
; For equality only:
    or   a               ; 4T   clear carry
    sbc  hl, de          ; 15T  HL = HL - DE, flags set
    add  hl, de          ; 11T  restore HL
                         ; --- 30T total, Z flag valid
```

Stack-Based Screen Fill

The fastest way to fill the screen with a pattern. See Chapter 3.

```
; fill_screen: fill 6144 bytes using PUSH
; Input:  HL = 16-bit fill pattern
; Cost:   ~36,000 T-states (vs ~129,000 with LDIR)
;
fill_screen:
    di                           ; 4T
    ld   (restore_sp + 1), sp    ; 20T  save SP (self-modifying)
    ld   sp, $5800               ; 10T  end of pixel area

    ld   b, 192                  ; 7T   192 iterations x 16 pushes x 2 bytes =
↪   6144
.loop:
    REPT 16
        push hl                  ; 11T  x 16 = 176T
    ENDR
    djnz .loop                   ; 13T/8T

restore_sp:
    ld   sp, $0000               ; 10T  self-modified
    ei                           ; 4T
    ret                          ; 10T
```

Fast Pixel-Row Iteration (Split Counters)

From Introspec's DOWN_HL analysis (Hype, 2020). Eliminates all conditional branching from the inner loop. Total cost for 192 rows: 2,343T vs 5,922T for naive DOWN_HL calls.

```
; iterate all 192 rows with minimal overhead
; HL starts at $4000
;
iterate_screen:
```

```
    ld   hl, $4000          ; 10T
    ld   c, 3               ; 7T   three thirds

.third:
    ld   b, 8               ; 7T   eight character rows per third

.char_row:
    push hl                 ; 11T  save char row start

    REPT 7
        ; ... process row using HL ...
        inc  h              ; 4T   next scanline (NO branching)
    ENDR
    ; ... process 8th row ...

    pop  hl                 ; 10T  restore char row start
    ld   a, l              ; 4T
    add  a, 32             ; 7T   next character row
    ld   l, a              ; 4T
    djnz .char_row         ; 13T/8T

    ld   a, h              ; 4T
    add  a, 8              ; 7T   next third
    ld   h, a              ; 4T
    dec  c                 ; 4T
    jr   nz, .third        ; 12T/7T
```

Quick Cost Comparisons

For inner loop decisions, these comparisons matter most:

Operation	Slow way	Fast way	Savings
Zero A	`LD A,0` (7T, 2B)	`XOR A` (4T, 1B)	3T, 1B
Test A=0	`CP 0` (7T, 2B)	`OR A` (4T, 1B)	3T, 1B
Copy 1 byte to mem	`LD (HL),A+INC HL` (13T)	`LDI` (16T)	LDI is slower but auto-increments DE too
Copy N bytes	`LDIR` (21T/byte)	N x `LDI` (16T/byte)	24% faster, costs 2N bytes of code
Fill 2 bytes	`LD (HL),A+INC HL` x2 (26T)	`PUSH rr` (11T)	58% faster, needs SP hijack
8-bit loop	`DEC B+JR NZ` (16T, 3B)	`DJNZ` (13T, 2B)	3T, 1B per iteration
Indirect call	`CALL nn` (17T, 3B)	`RET` via render list (10T, 1B)	7T, 2B per dispatch
Register swap	`LD A,H+LD H,D+LD D,A` (12T, 3B)	`EX DE,HL` (4T, 1B)	8T, 2B
Save 6 registers	3 x `PUSH` (33T, 3B)	`EXX` (4T, 1B)	29T, 2B

Instruction Encoding Size Reference

For size-coding and estimating code density:

Prefix	Instructions	Extra bytes	Extra T-states
None	Most 8-bit ops, LD, ADD, INC, PUSH, POP, JP, JR	0	0
CB	Bit ops, shifts, rotates on registers	+1	+4 typically
ED	Block ops, 16-bit ADC/SBC, IN/OUT (C), LD rr,(nn)	+1	varies
DD	IX-indexed operations	+1	+4 to +8
FD	IY-indexed operations	+1	+4 to +8
DD CB	Bit/shift/rotate on (IX+d)	+2	+8 to +12

Size-coding tip: Avoid IX/IY-indexed instructions when possible. `LD A,(IX+5)` is 3 bytes/19T. `LD L,5 / LD A,(HL)` is 3 bytes/11T if H already holds the page. The index registers are convenient but expensive.

Sources: Zilog Z80 CPU User Manual (UM0080); Sean Young, "The Undocumented Z80 Documented" (2005); Dark / X-Trade, "Programming Algorithms" (Spectrum Expert #01, 1997); Introspec, "Once more about DOWN_HL" (Hype, 2020); Chapter 1 (timing harness); Chapter 3 (toolbox patterns); Chapter 4 (multiply, division)

Appendix B: Sine Table Generation and Trigonometric Tables

"Cosine is just sine offset by a quarter period." – Raider's Commandments

---

Every demo effect that curves – rotation, plasma, scrolling, tunnels – needs a sine table. On the Z80, you pre-compute the values into a lookup table and index by angle. The question is how to store and access that table as efficiently as possible.

This appendix compares eight approaches to sine table storage, ranging from the obvious (256-byte table) to the exotic (2-bit second-order delta encoding). The data comes from `verify/sine_compare.py`, which you can run to reproduce every number here.

The Standard Format

A demoscene sine table has **256 entries**, indexed by angle:

Index	Angle
0	0°
64	90°
128	180°
192	270°
256 (wraps to 0)	360°

Each entry is a **signed byte** (-128 to +127), representing -1.0 to approximately +1.0. The power-of-two period means the angle index wraps naturally with 8-bit overflow, and cosine becomes sine by adding 64:

```
; sin(angle) -- direct table lookup
    ld   h, high(sin_table)  ; 7T   table must be 256-byte aligned
    ld   l, a                ; 4T   A = angle (0-255)
    ld   a, (hl)             ; 7T   A = sin(angle)
                             ; --- 18 T-states total

; cos(angle) -- offset by quarter period
    add  a, 64               ; 7T   cos = sin + 90°
```

```
    ld   l, a               ; 4T
    ld   a, (hl)            ; 7T
```

This is Raider's key rule: store H once with the table's high byte, then L is the angle and wraps freely.

Comparison of Approaches

#	Approach	Data	Code	Total	RAM	Max Error	RMS
1	Full table (256 bytes)	256	0	**256**	0	0	0.00
2	Quarter-wave table	65	21	**86**	0	0	0.00
3	Parabolic approximation	0	38	**38**	0	8	4.51
4	Quarter-wave + 2nd-order deltas	18	45	**63**	64	0	0.00
5	Bhaskara I approximation	0	~60	**~60**	0	1	0.49
5b	Bhaskara I + correction bitmap	1	~80	**~81**	0	0	0.00
6	Quarter-wave + packed 4-bit deltas	33	43	**76**	64	0	0.00
7	Full 2nd-order deltas, 2-bit packed	66	30	**96**	256	0	0.00

The approaches fall into three categories:

- **Lookup-based** (no RAM needed): full table, quarter-wave table
- **Generation-based** (needs RAM buffer at startup): delta and second-order delta approaches
- **Approximate** (no table at all): parabolic, Bhaskara I
- **Approximate + exact correction**: Bhaskara I with correction bitmap

---

Approach 1: Full 256-Byte Table

The simplest and fastest. Pre-compute all 256 values and embed them as data.

```
; Lookup: 18 T-states, zero error
    ld   h, high(sin_table)
    ld   l, a
    ld   a, (hl)
```

Cost: 256 bytes of ROM. **Speed:** 18 T-states per lookup. **When to use:** Always, unless you are size-coding. On a 48K Spectrum with ~40K free, 256 bytes is nothing. This is the default choice.

Approach 2: Quarter-Wave Table

A sine wave has four-fold symmetry. The first quadrant (0° to 90°, indices 0 to 64) contains all the information:

- **Second quadrant** (65-128): mirror of first quadrant. `sin(128 - i) = sin(i)`.
- **Third quadrant** (129-192): negative of first quadrant. `sin(128 + i) = -sin(i)`.
- **Fourth quadrant** (193-255): negative mirror. `sin(256 - i) = -sin(i)`.

Store only 65 bytes (indices 0 through 64 inclusive), then reconstruct:

```
; Quarter-wave sine lookup
; Input:  A = angle (0-255)
; Output: A = sin(angle), signed byte
; Uses:   HL, BC
; Table:  sin_quarter (65 bytes, 256-byte aligned)
;
qsin:
    ld   c, a               ; 4T   save original angle
    and  $7F                ; 7T   fold to 0-127 (first half)
    cp   65                 ; 7T   past the peak?
    jr   c, .no_mirror      ; 12/7T
    ; Mirror: index = 128 - index
    neg                     ; 8T
    add  a, 128             ; 7T
.no_mirror:
    ld   h, high(sin_quarter) ; 7T
    ld   l, a               ; 4T
    ld   a, (hl)            ; 7T   A = |sin(angle)|
    bit  7, c               ; 8T   was original angle >= 128?
    ret  z                  ; 11/5T  no: positive half, done
    neg                     ; 8T   yes: negate for third/fourth quadrant
    ret                     ; 10T
```

Cost: 65 bytes data + ~21 bytes code = **86 bytes total**. **Speed:** ~50-70 T-states per lookup (varies by quadrant). **Error:** Zero. **When to use:** Size-limited demos (256-byte, 512-byte intros) where you need exact values but cannot spare 256 bytes.

Approach 3: Parabolic Approximation (Dark's Method)

From Dark / X-Trade, *Spectrum Expert* #01 (1997). The idea: half a period of cosine looks like a parabola. The approximation $y \approx 1 - 2(x/\pi)^2$ matches closely. In integer terms, each half-period is generated as a piecewise quadratic.

Pure code, zero data. The generation loop needs an 8×8 multiply and some accumulator logic – approximately 38 bytes.

The error is bounded: **maximum absolute error = 8** (out of a 256-step range), or about **6.3%** of full scale. The RMS error is 4.51.

Here is where the parabola diverges from true sine (first quadrant):

```
Index  True  Para  Diff
    0     0     0    +0
    4    12    15    -3
    8    25    30    -5
   12    37    43    -6
   16    49    56    -7
   20    60    67    -7
   24    71    77    -6
   28    81    87    -6
   32    88    93    -5
```

The parabola is consistently "ahead" – it rises faster near zero and is flatter near the peak. Maximum divergence: **8 units at index 17** (about 24°).

When to use: Extreme size-coding (64-byte intros, tight loaders). Plasmas, simple scrollers, and wobble effects where the eye does not distinguish exact curvature from approximate. Not suitable for smooth rotation or precise wireframe 3D.

Approach 4: Second-Order Delta Encoding (The Deep Trick)

This is the most mathematically interesting approach, and by far the most compact exact representation.

The Key Insight

The second derivative of sin(x) is -sin(x). At 8-bit integer precision, quantising to signed bytes, the second finite difference of the sine table has a remarkable property: **every value is exactly -1, 0, or +1**.

```
True sine:      [0,  3,  6,  9, 12, 16, 19, 22, 25, ...]
First diff:     [3,  3,  3,  3,  4,  3,  3,  3,  3, ...]
Second diff:    [0,  0,  0,  1, -1,  0,  0,  0,  0, ...]
```

Three values. Two bits per entry. This is not an approximation – it is exact. The mathematical reason: $d^2(\sin)/dx^2$ is a smooth function with small magnitude, and at 256 entries per period with 8-bit amplitude, the discrete second derivative never exceeds ±1.

Full Table via 2-Bit Second-Order Deltas

Store: initial value (1 byte), initial delta (1 byte), then 254 second-order deltas packed at 2 bits each (64 bytes). **Total: 66 bytes data + ~30 bytes decode code = 96 bytes.** Needs 256 bytes of RAM to decode into.

Quarter-Wave via 2-Bit Second-Order Deltas

Combine with quarter-wave symmetry: store only the first 64 second-order deltas. **Total: 18 bytes data + ~45 bytes decode code = 63 bytes.** Needs 64 bytes of RAM.

This is the **smallest exact representation**: 63 bytes total for a perfect 256-entry sine table.

```
; Decode quarter-wave from 2-bit second-order deltas
; sin_d2_data: 16 bytes of packed 2-bit deltas (64 entries)
; sin_buffer:  64 bytes RAM for decoded quarter-wave
;
decode_quarter_d2:
    ld   hl, sin_buffer      ; destination
    ld   de, sin_d2_data     ; source (packed d2 values)
    xor  a
    ld   (hl), a             ; sin[0] = 0
    inc  hl
    ld   b, a                ; b = current delta (starts at 0)
    ld   c, 63               ; 63 more entries to decode

.loop:
    ; Unpack 2-bit d2 value
    ; 00 = 0, 01 = +1, 11 = -1 (10 unused)
    rr   (de)                ; shift out 2 bits
    rr   (de)
    ; ... (bit extraction logic)

    ; Apply: delta += d2, value += delta
    add  a, b                ; new delta
    ld   b, a
    ld   a, (hl-1)           ; previous value (pseudocode)
    add  a, b
    ld   (hl), a
    inc  hl
    dec  c
    jr   nz, .loop

    ; Now use qsin() lookup on sin_buffer
```

The decode runs once at startup. After that, use the quarter-wave lookup routine from Approach 2 on the decoded buffer.

When to use: Size-coded demos (128-byte, 256-byte intros) where you need exact values, can afford 64 bytes of RAM, and have a brief startup phase. The decode loop runs in under 2,000 T-states – invisible.

Sidebar: Why Not 1 Bit Per Delta?

An intuitive objection: the quarter-wave sine (0° to 90°) is monotonically increasing. First differences d1 are always non-negative. In continuous maths, the second derivative of sine in the first quadrant is always negative (the curve is concave). So d2 should be ≤ 0, meaning we only need {-1, 0} – a single bit per entry.

The intuition is right for continuous sine, but wrong for quantised integer sine. At 8-bit precision, rounding creates occasional upward bumps in d1:

```
d1:  3, 3, 3, 3, 4, 3, 3, ...  (that 4 is a rounding correction)
d2:  0, 0, 0, +1, -1, 0, ...   (the +1 is load-bearing)
```

There are 12 such +1 entries out of 63. If you suppress them (cap d1 to be monotonically non-increasing), the errors *accumulate*: by index 64 the peak reaches only 108 instead of 127 – a max error of 19, worse than the parabolic approximation. Those +1 corrections carry precisely the information needed to hit the right integer values. You cannot drop them.

A variable-length prefix code (0 → 1 bit, ±1 → 2 bits) saves 4 bytes of data over fixed 2-bit encoding but costs ~15 extra bytes of Z80 decode logic. Net loss. The 2-bit fixed encoding is the practical optimum.

Sidebar: Why Parabolic + Correction Doesn't Help

Another intuitive idea: generate a parabolic approximation (38 bytes of code, max error 8), then store a small correction table to fix it to exact values. The corrections range from -8 to +8, so they should compress well.

The corrections *do* compress well – their first differences are exactly {-1, 0, +1}, packing into 2 bits per entry. But this is no coincidence. A parabola is a quadratic with constant second derivative. So:

- `d2(sin)` ∈ {-1, 0, +1} – the sine's second derivative at integer precision
- `d2(para)` ∈ {-1, 0, +1} – the parabola's second derivative (nearly constant)
- `d1(correction)` = `d1(sin) - d1(para)` ∈ {-1, 0, +1} – **same entropy**

The correction deltas have *exactly the same structure* as the direct sine d2. But the parabolic route adds 38 bytes of generation code, plus ~20 bytes to apply corrections. Total: ~96 bytes vs 63 bytes for direct d2 encoding.

The parabola removes the smooth (low-frequency) component of sine – but the 2-bit d2 encoding already handles smooth data perfectly. There is nothing left for the parabola to contribute that d2 doesn't already capture. The generation code is pure overhead.

Approach 5: Bhaskara I Approximation (7th Century)

The most surprising entry in our comparison comes from 7th-century Indian mathematician Bhaskara I. His rational approximation to sine, published around 629 CE,

achieves **max error of only 1 unit** at 8-bit precision – dramatically better than the parabolic approximation (max error 8) and nearly exact.

The Formula

For angle x in radians (0 to π):

```
sin(x) ≈ 16x(π - x) / (5π² - 4x(π - x))
```

In our integer domain (angle 0-64 for the first quadrant, amplitude 0-127):

```
sin(i) ≈ 127 × 16i(64 - i) / (5 × 64² - 4 × i(64 - i))
       = 127 × 16i(64 - i) / (20480 - 4i(64 - i))
```

The formula is a ratio of two quadratics. On the Z80 this needs an 8×8 multiply and a 16-bit division – routines that many demos already include for 3D projection or texture mapping.

Accuracy

Across the 65 entries of the first quadrant, Bhaskara I matches the exact integer sine everywhere except **8 positions** (out of 65), where it is off by exactly ±1:

Index	True	Bhaskara	Diff
4	12	13	-1
17	51	52	-1
28	81	80	+1
31	88	87	+1
40	106	105	+1
43	111	110	+1
50	120	119	+1
52	122	121	+1

Only 8 positions differ, all by exactly ±1. The errors are split: 2 entries where Bhaskara overshoots (near the start), 6 where it undershoots (near the peak). Eight corrections total, which encode as a single byte bitmap.

Z80 Implementation

The implementation requires: - An 8×8→16 multiply routine (~20 bytes, likely already available) - A 16÷16→16 divide routine (~30 bytes, likely already available) - The Bhaskara wrapper itself (~25 bytes) - Quarter-wave folding logic (~15 bytes, shared with Approach 2)

If your demo already has multiply and divide routines, the marginal cost is roughly **25 bytes** for a sine function with max error 1.

If you need the routines from scratch, total is approximately **60 bytes** of code with zero data bytes. This is competitive with the d2 delta approach (63 bytes) but requires no RAM buffer and no startup decode phase. The tradeoff: 1 unit of error vs perfect accuracy.

Bhaskara I + Correction Bitmap (Exact)

To eliminate that last unit of error, store the 8 correction positions as a bitmap. Since the corrections are symmetric (first 4 need +1, last 4 need -1), one byte suffices:

```
; After computing Bhaskara approximation in A, index in C:
    push af
    ld   a, c
    ; Look up correction from bitmap (8 specific indices)
    ; ... (~20 bytes of correction logic)
    pop  af
    add  a, correction      ; ±1 or 0
```

Total: ~80 bytes code + 1 byte data = **~81 bytes**, zero RAM, zero startup, exact values. More expensive than d2 deltas (63B) but avoids the RAM buffer and startup decode.

When to Use Bhaskara I

- **You already have multiply/divide routines:** ~25 bytes extra, max error 1. Hard to beat.
- **No RAM available for decode buffer:** Unlike d2 deltas, Bhaskara computes on the fly.
- **Real-time generation needed:** Each value is computed independently – no sequential dependency, so you can compute sin(any angle) without decoding a table first.
- **±1 error is acceptable:** For scrollers, plasmas, and most visual effects, the difference between max error 1 and max error 0 is literally invisible.

  Historical Note: Bhaskara I's formula predates European trigonometric tables by nearly a millennium. That a 7th-century rational approximation achieves max error 1 on a 1980s 8-bit processor is a beautiful collision of mathematical elegance and engineering constraints. The formula was published in *Mahabhaskariya* (629 CE), a commentary on Aryabhata's astronomical methods.

Practical Recommendations

Every generation-based approach produces a lookup table at startup. After that, the runtime cost is identical: `LD H, high(table) / LD L, A / LD A, (HL)` = **18 T-states** for a 256-byte table, or the quarter-wave folding routine at **50-70 T-states** for a 64-byte buffer. The "ROM cost" column below is what matters for size-coding – it is the total bytes your approach occupies in the binary.

Use Case	Approach	ROM Cost	RAM	Init	Lookup	Error
Normal demo / game	Full 256-byte table	256B	0	none	18 T	exact

Use Case	Approach	ROM Cost	RAM	Init	Lookup	Error
512-byte intro	Quarter-wave table	86B	0	none	50-70 T	exact
256-byte intro	Quarter + d2 deltas	63B	64B	~2K T	50-70 T	exact
Has multiply/divide	Bhaskara I (generate to LUT)	~25B extra	256B	~80K T	18 T	±1 max
128-byte intro	Parabolic (generate to LUT)	38B	256B	~10K T	18 T	±8 max

The Decision Tree

1. **Do you have 256 bytes to spare?** Use the full table. Do not overthink it. `LD L,A / LD A,(HL)` at 18 T-states cannot be beaten.
2. **Size-limited but need accuracy?** Quarter-wave table at 86 bytes. No RAM needed, no startup phase. Lookup is 50-70 T-states (the folding logic).
3. **Extreme size limit, exact values needed?** Quarter-wave + second-order delta decoding at 63 bytes. Decode once at startup into a 64-byte quarter buffer, then use the same folding lookup.
4. **Already have multiply/divide?** Bhaskara I at ~25 bytes extra. Generate a full 256-byte LUT at startup, then enjoy 18 T-state lookups with max error 1.
5. **Extreme size limit, approximate OK?** Parabolic at 38 bytes, zero data. Generate to a 256-byte LUT at startup. Max error 8, good for plasmas and wobbles.

What Does Not Work

- **Parabolic + correction table** (123 bytes exact): worse than just using a quarter-wave table (86 bytes). The overhead of computing the parabola *and* looking up a correction defeats the purpose.
- **Delta + RLE** (100-219 bytes): sine deltas vary smoothly rather than repeating in runs. RLE is designed for data with long constant runs – sine is the wrong shape for it.
- **Full delta-encoded table** (152-271 bytes): uses *more* total bytes than the raw 256-byte table. Delta encoding only helps when deltas are significantly smaller than the original values; sine deltas are already bounded to ±4, but you still need 256 of them.

Raider's Commandments

In the Hype comments on Introspec's analysis of *Illusion*, veteran coder Raider distilled decades of collective wisdom into informal "commandments" for sine table design:

1. **256 entries per full period.** The angle index wraps with 8-bit overflow. No modular arithmetic needed.
2. **Signed bytes: -128 to +127.** Matches Z80 signed arithmetic.
3. **Page-align the table.** Place it at a 256-byte boundary so H is constant. `LD H,high(table)` once, then `LD L,angle / LD A,(HL)` forever.
4. **Cosine is sine + 64.** One `ADD A,64` instruction.
5. **Sine of (angle + 128) = -sine(angle).** `NEG` flips the sign. Use this for phase shifts.
6. **Do not compute sine at runtime** unless you are size-coding. A table lookup is always faster.
7. **Keep the amplitude as a power of two** (64, 127, 128) so multiplication is a shift.
8. **Quarter-wave symmetry** saves 75% storage when every byte matters.
9. **Test at the boundaries.** Index 0 should be exactly 0. Index 64 should be the maximum positive value (+127). Index 128 should be exactly 0. Index 192 should be the maximum negative value (-128 or -127, depending on your convention).

These rules reflect decades of experience. Follow them and your sine tables will be fast, small, and correct.

Reference: The Full 256-Byte Table

For convenience, here is the standard sine table (256 entries, signed, period = 256, amplitude ±127):

```
; 256-byte sine table, page-aligned
; sin(0) = 0, sin(64) = +127, sin(128) = 0, sin(192) = -128
;
    ALIGN 256
sin_table:
    DB    0,   3,   6,   9,  12,  16,  19,  22
    DB   25,  28,  31,  34,  37,  40,  43,  46
    DB   49,  51,  54,  57,  60,  63,  65,  68
    DB   71,  73,  76,  78,  81,  83,  85,  88
    DB   90,  92,  94,  96,  98, 100, 102, 104
    DB  106, 108, 109, 111, 112, 114, 115, 117
    DB  118, 119, 120, 121, 122, 123, 124, 124
    DB  125, 126, 126, 127, 127, 127, 127, 127
    DB  127, 127, 127, 127, 127, 127, 126, 126
    DB  125, 124, 124, 123, 122, 121, 120, 119
    DB  118, 117, 115, 114, 112, 111, 109, 108
    DB  106, 104, 102, 100,  98,  96,  94,  92
    DB   90,  88,  85,  83,  81,  78,  76,  73
    DB   71,  68,  65,  63,  60,  57,  54,  51
    DB   49,  46,  43,  40,  37,  34,  31,  28
    DB   25,  22,  19,  16,  12,   9,   6,   3
    DB    0,  -3,  -6,  -9, -12, -16, -19, -22
    DB  -25, -28, -31, -34, -37, -40, -43, -46
    DB  -49, -51, -54, -57, -60, -63, -65, -68
```

```
    DB  -71, -73, -76, -78, -81, -83, -85, -88
    DB  -90, -92, -94, -96, -98,-100,-102,-104
    DB -106,-108,-109,-111,-112,-114,-115,-117
    DB -118,-119,-120,-121,-122,-123,-124,-124
    DB -125,-126,-126,-127,-127,-127,-127,-127
    DB -128,-127,-127,-127,-127,-127,-126,-126
    DB -125,-124,-124,-123,-122,-121,-120,-119
    DB -118,-117,-115,-114,-112,-111,-109,-108
    DB -106,-104,-102,-100, -98, -96, -94, -92
    DB  -90, -88, -85, -83, -81, -78, -76, -73
    DB  -71, -68, -65, -63, -60, -57, -54, -51
    DB  -49, -46, -43, -40, -37, -34, -31, -28
    DB  -25, -22, -19, -16, -12,  -9,  -6,  -3
```

Copy, paste, assemble, use.

Sources: Dark / X-Trade “Programming Algorithms” (Spectrum Expert #01, 1997) for the parabolic approximation; Bhaskara I, *Mahabhaskariya* (629 CE) for the rational approximation; Raider (Hype comments, 2017) for sine table design principles; `verify/sine_compare.py` for comparative analysis

Appendix C: Compression Quick Reference

> *"The question is not whether to compress — it is which compressor to use, and when."* – Chapter 14

This appendix is a tear-out reference card for data compression on the ZX Spectrum. Chapter 14 covers the theory, the benchmark data, and the reasoning behind each recommendation. This appendix distils it into lookup tables and decision rules you can pin above your monitor.

All numbers come from Introspec's 2017 benchmark ("Data Compression for Modern Z80 Coding," Hype) unless otherwise noted. The test corpus was 1,233,995 bytes of mixed data: Calgary/Canterbury academic benchmarks, 30 ZX Spectrum graphics, 24 music files, and miscellaneous demo data.

Compressor Comparison Table

Compressor	Author	Compressed (bytes)	Ratio	Decompressor Size	Speed (T/byte)	Backwards	Notes
Exomizer 2	Magnus Lind	596,161	48.3%	~170 bytes	~250	Yes	Best ratio. Slow to decompress.
ApLib	Joergen Ibsen	606,833	49.2%	~199 bytes	~105	No	Good all-rounder.

Compressor	Author	Compressed (bytes)	Ratio	Decompressor Size	Speed (T/byte)	Backwards	Notes
Hrust 1	Alone Coder	613,602	49.7%	~150 bytes	~120	Yes	Relocatable stack depacker. Popular in Russian scene.
Pu-Crunch	Pasi Ojala	616,855	50.0%	~200 bytes	~140	No	Originally for C64.
Pletter 5	XL2S	635,797	51.5%	~120 bytes	~69	No	Fast + decent ratio.
MegaLZ	LVD / Introspec	636,910	51.6%	92 bytes (compact)	~98 (compact)	No	Optimal parser. Revived 2019 with new decompressors.
MegaLZ fast	LVD / Introspec	636,910	51.6%	234 bytes	~63	No	Fastest MegaLZ variant. Faster than 3x LDIR.

Compressor	Author	Compressed (bytes)	Ratio	Decompressor Size	Speed (T/byte)	Backwards	Notes
ZX0	Einar Saukas	~642,000*	~52%	~70 bytes	~100	Yes	Successor to ZX7. Optimal parser. Modern default.
ZX7	Einar Saukas	653,879	53.0%	**69 bytes**	~107	Yes	Tiny decompressor. The classic size-coder tool.
Bitbuster	Team Bomba	~660,000*	~53.5%	~90 bytes	~80	No	Simple. Good for first projects.
LZ4	Yann Collet (Z80 port)	722,522	58.6%	~100 bytes	**~34**	No	Fastest decompression. Byte-aligned tokens.

Compressor	Author	Compressed (bytes)	Ratio	Decompressor Size	Speed (T/byte)	Backwards	Notes
Hrum	Hrumer	~642,000*	~52%	~130 bytes	~110	No	Popular in Russian scene. Declared obsolete by Introspec.
ZX1	Einar Saukas	—	~51%	~80 bytes	~90	Yes	ZX0 variant. Slightly better ratio, slightly larger decompressor.
ZX2	Einar Saukas	—	~50%	~100 bytes	~85	Yes	Used in RED REDUX 256b intro (2025). Best ZXn ratio.

* Approximate. ZX0, Bitbuster, and Hrum were not in the original 2017 benchmark; values are estimated from independent tests on similar corpora.

Reading the table:

- **Ratio** = compressed size / original size. Lower is better.

- **Speed** = T-states per output byte during decompression. Lower is faster.
- **Decompressor Size** = bytes of Z80 code needed for the decompression routine. Lower is better for size-coded intros.
- **Backwards** = supports decompressing from end to start, allowing in-place decompression when source and destination overlap.

Decision Tree: Which Compressor?

Follow top-to-bottom. Take the first branch that matches your situation.

```
START
  |
  +-- Is this a 256-byte or 512-byte intro?
  |     YES --> ZX0 (70-byte decompressor) or custom RLE (<30 bytes)
  |
  +-- Is this a 1K or 4K intro?
  |     YES --> ZX0 (best ratio-to-decompressor-size)
  |
  +-- Do you need real-time streaming (decompress during playback)?
  |     YES --> LZ4 (~34 T/byte = 2+ KB per frame at 50fps)
  |
  +-- Do you need fast decompression between scenes?
  |     YES --> MegaLZ fast (~63 T/byte) or Pletter 5 (~69 T/byte)
  |
  +-- Is decompression speed irrelevant (one-time load at startup)?
  |     YES --> Exomizer (48.3% ratio, nothing beats it)
  |
  +-- Need a good balance of ratio and speed?
  |     YES --> ApLib (~105 T/byte, 49.2% ratio)
  |
  +-- Is the data mostly runs of identical bytes?
  |     YES --> Custom RLE (decompressor < 30 bytes, trivial)
  |
  +-- Is the data sequential animation frames?
  |     YES --> Delta-encode first, then compress with ZX0 or LZ4
  |
  +-- First project, want something simple?
        YES --> Bitbuster or ZX0 (both well-documented, easy to integrate)
```

Compressibility of Common ZX Spectrum Data Types

How well different data types compress, and tricks to improve the ratio.

Data Type	Raw Size	Typical ZX0 Ratio	Typical Exomizer Ratio	Notes
Screen pixels ($4000-$57FF)	6,144 bytes	40–60%	35–55%	Depends on image complexity. Black backgrounds compress well.
Attributes ($5800-$5AFF)	768 bytes	30–50%	25–45%	Often highly repetitive. Solid-colour areas compress to almost nothing.
Full screen (pixels + attrs)	6,912 bytes	40–58%	35–52%	Compress pixels and attributes separately for 5–10% better ratio.
Sine/cosine tables	256 bytes	60–75%	55–70%	Smooth curves compress well. Consider generation instead (Appendix B).

Data Type	Raw Size	Typical ZX0 Ratio	Typical Exomizer Ratio	Notes
Tile data (8x8 tiles)	varies	35–55%	30–50%	Reorder tiles by similarity for better ratio.
Sprite data	varies	45–65%	40–60%	Mask bytes hurt ratio. Store masks separately.
PT3 music data	varies	40–55%	35–50%	Pattern data is repetitive. Empty rows compress well.
AY register dumps	varies	30–50%	25–45%	Highly repetitive between frames. Delta-encode first.
Lookup tables (arbitrary)	varies	50–80%	45–75%	Random-looking data compresses poorly. Presort if possible.

Data Type	Raw Size	Typical ZX0 Ratio	Typical Exomizer Ratio	Notes
Font data (96 chars x 8 bytes)	768 bytes	55–70%	50–65%	Lots of zero bytes (de-scen-ders, thin strokes).

Pre-compression tricks

These techniques improve compression ratio by restructuring data before feeding it to the compressor.

Separate pixels from attributes. A full 6,912-byte screen stored as one block forces the compressor to handle a transition from pixel data to attribute data at byte 6,144. Compress the 6,144-byte pixel block and the 768-byte attribute block separately. The attribute block, being highly repetitive, often compresses to under 200 bytes.

Delta-encode animation frames. Store the first frame in full. For each subsequent frame, store only the bytes that differ from the previous frame as (offset, value) pairs. Apply LZ compression to the delta stream. psndcj compressed 122 frames (843,264 bytes raw) into 10,512 bytes using this technique in Break Space.

Reorder data for locality. Tile maps stored in row-major order may compress better if reordered so that similar tiles are adjacent. Sort sprite frames by visual similarity. Group repeated sub-patterns together.

Store constants separately. If a data block contains a repeated header or footer (e.g., tile metadata), factor it out and store it once. Only compress the variable portion.

Interleave planes. For multicolour or masked sprites, storing all mask bytes together and all pixel bytes together often compresses better than interleaving mask-pixel-mask-pixel per row.

Minimal RLE Decompressor

The simplest useful compressor. Under 30 bytes. Suitable for 256-byte intros or data with long runs of identical bytes. See Chapter 14 for a full discussion.

```
; Minimal RLE decompressor
; Format: [count][value] pairs, terminated by count = 0
; HL = source (compressed data)
; DE = destination (output buffer)
; Destroys: AF, BC
rle_decompress:
        ld      a, (hl)         ; read count            7T
        inc     hl              ;                        6T
```

```
        or      a               ; count = 0?            4T
        ret     z               ; yes: done             5T/11T
        ld      b, a            ; B = count             4T
        ld      a, (hl)         ; read value            7T
        inc     hl              ;                       6T
.fill:  ld      (de), a         ; write value           7T
        inc     de              ;                       6T
        djnz    .fill           ; loop B times          13T/8T
        jr      rle_decompress  ; next pair             12T
; Total: 23 bytes of code
; Speed: ~26 T-states per output byte (within runs)
```

Encoding tool (Python one-liner for simple RLE):

```
def rle_encode(data):
    out = bytearray()
    i = 0
    while i < len(data):
        val = data[i]
        count = 1
        while i + count < len(data) and data[i + count] == val and count < 255:
            count += 1
        out.extend([count, val])
        i += count
    out.extend([0])  # terminator
    return out
```

This naive RLE expands data with no runs (worst case: 2 bytes per 1 byte of input). For mixed data, use escape-byte RLE: a special byte signals a run, and all other bytes are literals. Or just use ZX0.

ZX0 Standard Decompressor (Z80)

The full standard forward decompressor by Einar Saukas. Approximately 70 bytes. This is the version you will use in most projects.

```
; ZX0 decompressor - standard forward version
; (c) Einar Saukas, based on Wikipedia description of LZ format
; HL = source (compressed data)
; DE = destination (output buffer)
; Destroys: AF, BC, DE, HL
dzx0_standard:
        ld      bc, $ffff       ; initial offset = -1
        push    bc              ; store offset on stack
        inc     bc              ; BC = 0 (literal length will be read)
        ld      a, $80          ; init bit buffer with end marker
dzx0s_literals:
        call    dzx0s_elias     ; read number of literals
        ldir                    ; copy literals from source to dest
        add     a, a            ; read next bit: 0 = last offset, 1 = new offset
        jr      c, dzx0s_new_offset
        ; reuse last offset
```

```
        call    dzx0s_elias     ; read match length
dzx0s_copy:
        ex      (sp), hl        ; swap: HL = offset, stack = source
        push    hl              ; put offset back on stack
        add     hl, de          ; HL = dest + offset = match source address
        ldir                    ; copy match
        add     a, a            ; read next bit: 0 = literal, 1 = match/offset
        jr      nc, dzx0s_literals
        ; new offset
dzx0s_new_offset:
        call    dzx0s_elias     ; read offset MSB (high bits)
        ex      af, af'         ; save bit buffer
        dec     b               ; B = $FF (offset is negative)
        rl      c               ; C = offset MSB * 2 + carry
        inc     c               ; adjust
        jr      z, dzx0s_done   ; offset = 256 means end of stream
        ld      a, (hl)         ; read offset LSB
        inc     hl
        rra                     ; LSB bit 0 -> carry = length bit
        push    bc              ; save offset MSB
        ld      b, 0
        ld      c, a            ; C = offset LSB >> 1
        pop     af              ; A = offset MSB (from push bc)
        ld      b, a            ; BC = full offset (negative)
        ex      (sp), hl        ; store offset, retrieve source
        push    bc              ; store offset again
        ld      bc, 1           ; minimum match length = 1
        jr      nc, dzx0s_copy  ; if carry clear: length = 1
        call    dzx0s_elias     ; otherwise read match length
        inc     bc              ; +1
        jr      dzx0s_copy
dzx0s_done:
        pop     hl              ; clean stack
        ex      af, af'         ; restore flags
        ret
; Elias interlaced code reader
dzx0s_elias:
        inc     c               ; C starts at 1
dzx0s_elias_loop:
        add     a, a            ; read bit
        jr      nz, dzx0s_elias_nz
        ld      a, (hl)         ; refill bit buffer
        inc     hl
        rla                     ; shift in carry
dzx0s_elias_nz:
        ret     nc              ; stop bit (0) = done
        add     a, a            ; read data bit
        jr      nz, dzx0s_elias_nz2
        ld      a, (hl)         ; refill
        inc     hl
        rla
dzx0s_elias_nz2:
```

```
        rl      c               ; shift bit into C
        rl      b               ; and into B
        jr      dzx0s_elias_loop
```

Usage:

```
        ld      hl, compressed_data     ; source address
        ld      de, $4000               ; destination (e.g., screen)
        call    dzx0_standard           ; decompress
```

Backwards variant. ZX0 also provides a backwards decompressor (`dzx0_standard_back`) that reads compressed data from end to start and writes output from end to start. This enables in-place decompression: place the compressed data at the end of the destination buffer, and decompress backwards so the output overwrites the compressed data only after it has been read. Essential when RAM is tight.

Integration Patterns

Pattern 1: Decompress to screen at startup

The most common use case. Load a compressed loading screen and display it.

```
        org     $8000
start:
        ld      hl, compressed_screen
        ld      de, $4000               ; screen memory
        call    dzx0_standard
        ; screen is now visible
        ; ... continue with demo/game ...

        include "dzx0_standard.asm"

compressed_screen:
        incbin  "screen.zx0"
```

Pattern 2: Decompress to buffer between effects

Decompress the next effect's data into a scratch buffer while the current effect is still running, or during a fade-out.

```
; During scene transition:
        ld      hl, scene2_data_zx0
        ld      de, scratch_buffer      ; e.g., $C000 in bank 1
        call    dzx0_standard
        ; scratch_buffer now holds the uncompressed data
        ; switch to scene 2, which reads from scratch_buffer
```

Pattern 3: Stream decompression during playback

For real-time effects that need a continuous data feed. LZ4 is the only practical choice here.

```
; Each frame: decompress next chunk
frame_loop:
        ld      hl, (lz4_read_ptr)     ; current position in compressed stream
        ld      de, frame_buffer
        ld      bc, 2048                ; bytes to decompress this frame
        call    lz4_decompress_partial
        ld      (lz4_read_ptr), hl     ; save position for next frame
        ; render from frame_buffer
        ; ...
        jr      frame_loop
```

At ~34 T/byte, LZ4 decompresses 2,048 bytes in 69,632 T-states — fitting within one frame (69,888 T-states on 48K). This is tight. Use border-time decompression or double-buffering for safety.

Pattern 4: Bank-switched compressed data (128K)

Store compressed data across multiple 16KB banks. Decompress from the currently paged bank, then switch banks when you run out.

```
; Page in bank containing compressed data
        ld      a, (current_bank)
        or      $10                     ; bit 4 = ROM select
        ld      bc, $7ffd
        out     (c), a                  ; page bank into $C000-$FFFF

        ld      hl, $C000              ; compressed data starts at bank base
        ld      de, dest_buffer
        call    dzx0_standard

        ; Page next bank for next asset
        ld      a, (current_bank)
        inc     a
        ld      (current_bank), a
```

For large demos with many compressed assets, maintain a table of (bank, offset, destination) tuples and loop through them during loading.

Build Pipeline: From Asset to Binary

The compression step belongs in your Makefile, not in your head.

```
Source asset       Converter        Compressor        Assembler
  (PNG)       -->   (png2scr)   -->   (zx0)      -->  (sjasmplus)  --> .tap
  (WAV)       -->   (pt3tools)  -->   (zx0)      -->  (incbin)
  (TMX)       -->   (tmx2bin)   -->   (exomizer)
```

Makefile rules:

```
# Compress .scr screens with ZX0
%.zx0: %.scr
    zx0 $< $@
```

```
# Compress large assets with Exomizer (one-time load)
%.exo: %.bin
    exomizer raw -c $< -o $@

# Build final binary
demo.bin: main.asm assets/title.zx0 assets/font.zx0
    sjasmplus main.asm --raw=$@
```

Tool installation:

Tool	Source	Install
ZX0	github.com/einar-saukas/ZX0	`gcc -O2 -o zx0 src/zx0.c src/compress.c src/optimize.c src/memory.c`
Exomizer	github.com/bitmanipulators/exomizer	`make` in `src/` directory
LZ4	github.com/lz4/lz4	`make` or `brew install lz4`
MegaLZ	github.com/Antonio-Cerra/megalzR	Older; check Introspec's Hype article for links

---

Quick Formulas

Bytes per frame at 50fps with decompressor X:

```
bytes_per_frame = 69,888 / speed_t_per_byte
```

Compressor	T/byte	Bytes/frame (48K)	Bytes/frame (128K Pentagon)
LZ4	34	2,055	2,108
MegaLZ fast	63	1,109	1,138
Pletter 5	69	1,012	1,038
ZX0	100	698	716
ApLib	105	665	682
Hrust 1	120	582	597
Exomizer	250	279	286

(128K Pentagon frame = 71,680 T-states)

Memory saved by compression over N screens:

```
saved = N * 6912 * (1 - ratio)
```

Example: 8 loading screens with Exomizer at 48.3% ratio save 8 * 6912 * 0.517 = 28,575 bytes — nearly two full 16KB banks.

See Also

- **Chapter 14:** Full discussion of compression theory, Introspec's benchmark, ZX0 internals, and the delta + LZ pipeline.

- **Appendix B:** Sine table generation — when tables are small enough, consider generating instead of compressing.
- **Appendix A:** Z80 instruction reference — LDIR, PUSH/POP, and other instructions used in decompressors.

Sources: Introspec “Data Compression for Modern Z80 Coding” (Hype, 2017); Introspec “Compression on the Spectrum: MegaLZ” (Hype, 2019); Einar Saukas, ZX0/ZX7/ZX1/ZX2 (github.com/einar-saukas); Break Space NFO (Thesuper, 2016)

Appendix D: Development Environment Setup

> *"You need five things: an editor, an assembler, an emulator, a debugger, and a Makefile. Everything else is optional."* – Chapter 1

This appendix walks you through setting up a complete Z80 development environment from scratch. By the end, you will be able to compile every assembly example in this book, run them in an emulator, and debug them with breakpoints and register inspection. The instructions cover macOS, Linux, and Windows.

If you have already followed the setup in Chapter 1, you have most of this in place. This appendix adds detail, covers alternative configurations, and serves as a single reference you can return to when setting up a new machine.

---

1. The Assembler: sjasmplus

Every code example in this book is written for **sjasmplus**, an open-source Z80/Z80N macro assembler by z00m128. It supports the full Z80 instruction set including all IX/IY indexed modes, macros, Lua scripting, multiple output formats, and expressions that make demoscene code practical to write.

Installing from Source

The most reliable way to get sjasmplus is to build it from source. This guarantees you have a known version and avoids platform-specific packaging issues.

```
git clone https://github.com/z00m128/sjasmplus.git
cd sjasmplus
make
```

On macOS, you need Xcode command-line tools (`xcode-select --install`). On Linux, you need `g++` and `make` (install via your package manager). On Windows, use MinGW or WSL.

After building, copy the `sjasmplus` binary to somewhere in your PATH:

```
# macOS / Linux
sudo cp sjasmplus /usr/local/bin/

# Verify
sjasmplus --version
```

You should see version 1.20.x or later. This book was developed and tested with v1.21.1.

Version Pinning

The book's repository pins sjasmplus as a git submodule in `tools/sjasmplus/`. If you clone the repository with `--recursive`, you get the exact version used to compile every example:

```
git clone --recursive https://github.com/[repo]/antique-toy.git
cd antique-toy/tools/sjasmplus
make
```

This is the safest approach. Assembler behaviour can change between versions – an expression that works in 1.21 might parse differently in 1.22.

Key Flags

Flag	Purpose	Example
`--nologo`	Suppress the startup banner	`sjasmplus --nologo main.a80`
`--raw=FILE`	Output a raw binary (no header)	`sjasmplus --raw=output.bin main.a80`
`--sym=FILE`	Write a symbol file (for debuggers)	`sjasmplus --sym=output.sym main.a80`
`--fullpath`	Show full file paths in error messages	Useful with VS Code problem matcher
`--msg=war`	Suppress info messages, show only warnings and errors	Cleaner build output

A typical build command for a chapter example:

```
sjasmplus --nologo --raw=build/example.bin
↪  chapters/ch01-thinking-in-cycles/examples/timing.a80
```

File Extension

All Z80 assembly source files in this book use the `.a80` extension. This is a convention, not a requirement – sjasmplus does not care about extensions. We use `.a80` because it is recognised by the Z80 Macro Assembler VS Code extension and distinguishes our source from other assembly dialects.

Hex Notation

The book uses `$FF` for hexadecimal values. sjasmplus also accepts `#FF` and `0FFh`, but `$FF` is the convention throughout. The standalone `$` symbol represents the current program counter address, used in constructs like `jr $` (infinite loop) or `dw $ + 4`.

---

2. The Editor: VS Code

Any text editor works. We recommend **Visual Studio Code** because of its Z80-specific extensions and integrated terminal. The entire workflow – edit, build, debug – happens in one window.

Essential Extensions

Install these from the VS Code Extensions marketplace (Ctrl+Shift+X):

Extension	Author	What It Does
Z80 Macro Assembler	mborik (`mborik.z80-macroasm`)	Syntax highlighting, code completion, symbol resolution for Z80 assembly. Understands sjasmplus syntax including macros and local labels.
Z80 Assembly Meter	Nestor Sancho	Displays byte count and T-state cost of selected instructions in the status bar. Select a block, see its total cost instantly. Indispensable for cycle counting.
DeZog	Maziac	Z80 debugger. Connects to emulators or its internal simulator. Breakpoints, register watches, memory inspection. See Section 4.

Build Task

Set up a build task so Ctrl+Shift+B compiles your current file. Create `.vscode/tasks.json` in your project root:

```
{
  "version": "2.0.0",
  "tasks": [
    {
      "label": "Assemble Z80",
      "type": "shell",
      "command": "sjasmplus",
      "args": [
        "--fullpath",
        "--nologo",
        "--msg=war",
        "${file}"
      ],
      "group": {
        "kind": "build",
        "isDefault": true
      },
      "problemMatcher": {
```

```
        "owner": "z80",
        "fileLocation": "absolute",
        "pattern": {
          "regexp": "^(.*)\\((\\d+)\\):\\s+(error|warning):\\s+(.*)$",
          "file": 1,
          "line": 2,
          "severity": 3,
          "message": 4
        }
      }
    }
  ]
}
```

The `problemMatcher` parses sjasmplus error output so that clicking an error in the terminal jumps to the offending line. The `--fullpath` flag ensures file paths are absolute, which VS Code needs to resolve them correctly.

Recommended Settings

Add these to your workspace `.vscode/settings.json` for a clean Z80 editing experience:

```
{
  "editor.tabSize": 8,
  "editor.insertSpaces": false,
  "editor.rulers": [80],
  "files.associations": {
    "*.a80": "z80-macroasm"
  }
}
```

Tab size 8 with real tabs matches the traditional assembly convention where mnemonics and operands align at fixed columns.

---

3. Emulators

You need an emulator to run your assembled code. Different emulators serve different purposes.

ZEsarUX – Feature-Rich Debugging

ZEsarUX by cerebellio is the most feature-rich open-source ZX Spectrum emulator. It supports the full range of Spectrum models (48K, 128K, +2, +3, Pentagon, Scorpion, ZX-Uno, ZX Spectrum Next), has a built-in debugger, and integrates with DeZog for VS Code debugging.

Install:

- macOS: `brew install zesarux`
- Linux: Build from source or use the AppImage from https://github.com/chernandezba/zesarux

- Windows: Download the installer from the ZEsarUX website

Why ZEsarUX for this book: Most chapter examples target the ZX Spectrum 128K. ZEsarUX emulates 128K memory banking, AY sound, TurboSound (two AY chips), and the contention patterns of different models. Its built-in debugger shows registers, memory, and disassembly without needing VS Code. And its DeZog integration provides the full VS Code debugging experience when you need it.

Quick launch:

```
# Run a .sna snapshot
zesarux --machine 128k --nosplash output.sna

# Run a .tap file
zesarux --machine 128k --nosplash output.tap
```

Fuse – Lightweight and Accurate

Fuse (the Free Unix Spectrum Emulator) by Philip Kendall is lightweight, cycle-accurate, and available on every platform. It is the best choice for quick testing when you do not need the full debugger.

Install:

- macOS: `brew install fuse-emulator`
- Linux: `apt install fuse-emulator-sdl` (Debian/Ubuntu) or `dnf install fuse-emulator` (Fedora)
- Windows: Download from the Fuse website

Quick launch:

```
# Run with Pentagon timing (matches the book's T-state counts)
fuse --machine pentagon output.sna

# Run as 128K Spectrum
fuse --machine 128 output.tap
```

Fuse is particularly good for testing timing-sensitive code because its cycle accuracy is well-verified. If your border stripe timing harness (Chapter 1) shows different results in Fuse versus another emulator, trust Fuse.

Unreal Speccy – Windows, Pentagon-Focused

If you develop primarily on Windows and target Pentagon timing, **Unreal Speccy** is a strong choice. It has a built-in debugger with memory map, breakpoints, and AY register monitoring. It emulates Pentagon and Scorpion hardware accurately.

Download from http://dlcorp.nedopc.com/viewforum.php?f=27 or search for "Unreal Speccy Portable."

For Agon Light 2

The Agon Light 2 uses an eZ80 CPU and a different hardware architecture. Chapter 22 covers Agon development in detail. For emulation, **Fab Agon Emulator** provides a software simulation of the Agon hardware (eZ80 + ESP32 VDP). It is available

at https://github.com/tomm/fab-agon-emulator and runs on macOS, Linux, and Windows.

Which Emulator Should I Use?

Situation	Recommended Emulator
Day-to-day development, running examples	Fuse (fast startup, accurate)
Debugging with breakpoints and register watches	ZEsarUX + DeZog
AY/TurboSound music development	ZEsarUX (best AY emulation)
Pentagon timing verification	Fuse or Unreal Speccy
Agon Light 2 development	Fab Agon Emulator
Quick sanity check on Windows	Unreal Speccy

---

4. The Debugger: DeZog

DeZog by Maziac is a VS Code extension that turns your editor into a Z80 debugger. It connects to ZEsarUX, CSpect, or its own internal Z80 simulator and provides the debugging experience modern developers expect: breakpoints, stepping, register watches, memory inspection, disassembly view, and call stack.

Chapter 23 discusses DeZog in the context of AI-assisted development. This section covers the practical setup.

Installation

1. Open VS Code.
2. Go to Extensions (Ctrl+Shift+X).
3. Search for “DeZog” by Maziac.
4. Click Install.

Connecting to ZEsarUX

DeZog communicates with ZEsarUX over a socket connection. First, launch ZEsarUX with its ZRCP (ZEsarUX Remote Control Protocol) server enabled:

```
zesarux --machine 128k --enable-remoteprotocol --remoteprotocol-port 10000
```

Then create a `.vscode/launch.json` in your project:

```
{
  "version": "0.2.0",
  "configurations": [
    {
```

```
      "type": "dezog",
      "request": "launch",
      "name": "DeZog + ZEsarUX",
      "remoteType": "zesarux",
      "zesarux": {
        "port": 10000
      },
      "sjasmplus": [
        {
          "path": "build/output.sld"
        }
      ],
      "topOfStack": "$FF00",
      "commandsAfterLaunch": [
        "-sprites disable",
        "-patterns disable"
      ]
    }
  ]
}
```

The `sjasmplus` section points to the `.sld` (Source Level Debug) file that sjasmplus generates with the `--sld=FILE` flag. This gives DeZog source-level debugging – breakpoints on source lines, not just addresses.

To generate the `.sld` file, add the flag to your build command:

```
sjasmplus --nologo --raw=build/output.bin --sld=build/output.sld
↪  --sym=build/output.sym main.a80
```

Using the Internal Simulator

For quick debugging without launching an external emulator, DeZog includes a built-in Z80 simulator. Change your `launch.json` to:

```
{
  "type": "dezog",
  "request": "launch",
  "name": "DeZog Internal Simulator",
  "remoteType": "zsim",
  "zsim": {
    "machine": "spectrum",
    "memoryModel": "ZX128K"
  },
  "sjasmplus": [
    {
      "path": "build/output.sld"
    }
  ],
  "topOfStack": "$FF00"
}
```

The internal simulator is faster to start and does not require ZEsarUX to be installed. It lacks accurate contention emulation, so do not use it for timing-critical debugging

– but for logic debugging (does my multiply routine produce the right result?), it is perfect.

Key DeZog Features

Breakpoints. Click the gutter next to a source line to set a breakpoint. Execution pauses when the Z80 program counter reaches that address. You can also set conditional breakpoints (e.g., break when `A == $FF`).

Register watches. The Variables panel shows all Z80 registers: AF, BC, DE, HL, IX, IY, SP, PC, and the alternate set (AF', BC', DE', HL'). Individual flags (C, Z, S, P/V, H, N) are broken out for easy reading.

Memory inspection. The Memory panel shows a hex dump of any address range. You can type an address and see what is there. Essential for verifying lookup tables, screen memory contents, and stack state.

Disassembly view. Even without source-level debugging, DeZog disassembles the code around the current PC. Useful for understanding what self-modifying code actually looks like at runtime.

Call stack. DeZog tracks CALL/RET pairs and shows a call stack. This works for conventional code. Self-modifying code and RET-chaining (Chapter 3) will confuse the stack tracker – this is expected.

---

5. Building the Book's Examples

Clone the Repository

```
git clone --recursive https://github.com/[repo]/antique-toy.git
cd antique-toy
```

The `--recursive` flag pulls the sjasmplus submodule in `tools/sjasmplus/`. If you already cloned without it:

```
git submodule update --init --recursive
```

Prerequisites

You need `make` and a C++ compiler (for building sjasmplus from the submodule). On most systems these are already present:

- macOS: `xcode-select --install`
- Debian/Ubuntu: `sudo apt install build-essential`
- Fedora: `sudo dnf install make gcc-c++`
- Windows: Use WSL, or install MinGW and GNU Make

Build Commands

The project Makefile handles everything. All compiled output goes to `build/`, which is gitignored.

Command	What It Does
`make`	Compile all chapter examples with sjasmplus
`make test`	Assemble all examples, report pass/fail for each
`make ch01`	Compile only Chapter 1 examples
`make ch04`	Compile only Chapter 4 examples
`make demo`	Build the "Antique Toy" companion demo
`make clean`	Remove all build artifacts

A successful `make test` run looks like this:

```
  OK  chapters/ch01-thinking-in-cycles/examples/timing.a80
  OK  chapters/ch04-maths/examples/multiply.a80
  OK  chapters/ch05-wireframe-3d/examples/cube.a80
  ...
---
12 passed, 0 failed
```

If any example fails, the output shows `FAIL` with the filename. Run the failing file manually with sjasmplus to see the detailed error:

```
sjasmplus --nologo chapters/ch04-maths/examples/multiply.a80
```

Running an Example

After compiling, load the output binary into your emulator. The exact method depends on the output format:

```
# If the example produces a .sna snapshot
fuse --machine pentagon build/ch01-thinking-in-cycles/examples/timing.sna

# If you built a raw binary, you need to create a .tap or .sna first,
# or load it at the correct address in the emulator's debugger
```

Most chapter examples use ORG `$8000` and produce raw binaries. To run them, either:

1. Use a `.tap` wrapper (the Makefile generates these when the source includes the appropriate directives), or
2. Load the binary at address `$8000` in your emulator's debugger and set PC to `$8000`.

---

6. Project Structure for Your Own Code

When you start writing your own Z80 code beyond the book's examples, here is a recommended directory layout:

```
my-demo/
  src/
    main.a80          -- entry point, includes other files
    effects/
      plasma.a80      -- individual effect routines
```

```
    scroller.a80
  data/
    font.a80        -- DB/DW data tables
    sintable.a80
  lib/
    multiply.a80    -- reusable utility routines
    draw_line.a80
assets/
  title.scr         -- raw screen files
  music.pt3         -- tracker music
build/              -- compiled output (gitignored)
Makefile
.vscode/
  tasks.json        -- build task
  launch.json       -- DeZog configuration
```

Minimal Makefile

```
SJASMPLUS ?= sjasmplus
BUILD_DIR := build
SJASM_FLAGS := --nologo

.PHONY: all clean run

all: $(BUILD_DIR)/demo.bin

$(BUILD_DIR)/demo.bin: src/main.a80 src/effects/*.a80 src/data/*.a80
↪  src/lib/*.a80
    @mkdir -p $(BUILD_DIR)
    $(SJASMPLUS) $(SJASM_FLAGS) --raw=$@ --sym=$(BUILD_DIR)/demo.sym
    ↪  --sld=$(BUILD_DIR)/demo.sld $<

run: $(BUILD_DIR)/demo.bin
    fuse --machine 128 $(BUILD_DIR)/demo.sna

clean:
    rm -rf $(BUILD_DIR)
```

Key points:

- The main source file uses `INCLUDE` directives to pull in other files. sjasmplus resolves includes relative to the source file's directory.
- The `--sym` flag generates a symbol file for reference. The `--sld` flag generates source-level debug data for DeZog.
- List all included source files as dependencies so `make` rebuilds when any file changes.

Include Convention

In your `main.a80`:

```
    ORG $8000
```

```
    ; --- Entry point ---
start:
    di
    ; set up stack, interrupts, etc.
    ld   sp, $FF00
    ; ...

    include "lib/multiply.a80"
    include "effects/plasma.a80"
    include "data/sintable.a80"
```

sjasmplus processes `INCLUDE` files inline, as if the text were pasted at that point. Keep your includes organised: library routines first, then effects, then data (since data is typically placed at the end of the binary).

---

7. Alternative Tools

This book uses sjasmplus because it is the most capable Z80 assembler for demoscene and game development work. But you may encounter other tools in the community.

Other Assemblers

Assembler	Notes
z80asm	Part of the z88dk C cross-compiler suite. Good if you mix C and assembly. Different syntax conventions.
RASM	By Roudoudou. Fast, supports CPC and Spectrum. Popular in the Amstrad CPC scene.
pasmo	Simple, portable, limited features. Suitable for small standalone programs but lacks macros and advanced features needed for larger projects.

The book's examples use sjasmplus-specific features (local labels with `.`, negative DB values, `$` hex prefix) that may not work unmodified in other assemblers. If you want to port an example to a different assembler, the changes are usually minor: label syntax, hex notation, and directive names.

Other VS Code Extensions

Extension	Notes
SjASMPlus Syntax	Alternative syntax highlighting tuned specifically for sjasmplus. Try it if `z80-macroasm` does not highlight a sjasmplus-specific feature correctly.
Z80 Debugger (Spectron)	An older VS Code Z80 debugger. DeZog has largely superseded it.

CSpect – Next-Focused Emulator

If you have a ZX Spectrum Next or are targeting Next-specific features (copper, layer 2, DMA, 28MHz turbo mode), **CSpect** by Mike Dailly is the reference emulator. DeZog connects to CSpect the same way it connects to ZEsarUX. CSpect is Windows-only but runs under Wine on macOS and Linux.

SpectrumAnalyzer

A browser-based tool that visualises ZX Spectrum screen memory layout, attribute conflicts, and timing. Useful for understanding Chapter 2's discussion of the screen interleave. Available at https://shiru.untergrund.net/spectrumanalyzer.html (or search for "ZX Spectrum screen analyzer").

---

8. Troubleshooting

"sjasmplus: command not found"

The binary is not in your PATH. Either copy it to `/usr/local/bin/` (macOS/Linux), add its directory to your PATH, or set the `SJASMPLUS` variable when running make:

```
make SJASMPLUS=/path/to/sjasmplus
```

Compilation errors in the book's examples

First, make sure you are using the sjasmplus version from the submodule (`tools/sjasmplus/`). Newer or older versions may have different behaviour. Second, check that you are assembling the file from the correct directory – sjasmplus resolves `INCLUDE` paths relative to the source file, not the working directory.

DeZog cannot connect to ZEsarUX

1. Make sure ZEsarUX is running with `--enable-remoteprotocol`.
2. Check that the port number in `launch.json` matches the `--remoteprotocol-port` argument.
3. On macOS, you may need to allow ZEsarUX through the firewall (System Settings > Privacy & Security).
4. Try restarting ZEsarUX before launching the DeZog session.

Emulator shows garbled screen

If you load a raw binary and see garbage, the most likely cause is a wrong load address. The book's examples use ORG `$8000`. Make sure your emulator loads the binary at that address, not at `$0000` or some other default. Using `.sna` or `.tap` output (which includes address information) avoids this problem.

Build output is empty or zero bytes

Check that your source file has an `ORG` directive and at least one instruction. sjasmplus with `--raw=` produces a binary from the first byte emitted to the last. If nothing is emitted (e.g., the file contains only `ORG $8000` with no code), the output is empty.

Tool Reference

Tool	Purpose	Official URL	Book Reference
sjasmplus	Z80/Z80N macro assembler	https://github.com/z00m128/sjasmplus	Chapter 1, all examples
VS Code	Editor and IDE	https://code.visualstudio.com	Chapter 1
Z80 Macro Assembler	VS Code syntax extension	Marketplace: `mborik.z80-macroasm`	Chapter 1
Z80 Assembly Meter	Cycle count display	Marketplace: Nestor Sancho	Chapter 1
DeZog	VS Code Z80 debugger	Marketplace: Maziac / https://github.com/maziac/DeZog	Chapter 23
ZEsarUX	Feature-rich Spectrum emulator	https://github.com/chernandezba/zesarux	Chapter 1, Chapter 23
Fuse	Lightweight Spectrum emulator	https://fuse-emulator.sourceforge.net	Chapter 1
Unreal Speccy	Pentagon-focused emulator	http://dlcorp.nedopc.com	Chapter 1
CSpect	ZX Spectrum Next emulator	https://dailly.blogspot.com	Chapter 22 (Next features)
Fab Agon Emulator	Agon Light 2 emulator	https://github.com/tomm/fab-agon-emulator	Chapter 22
ZX0	Optimal LZ compressor	https://github.com/einar-saukas/ZX0	Chapter 14, Appendix C
Exomizer	Best-ratio compressor	https://github.com/bitmanipulators/exomizer	Chapter 14, Appendix C

Tool	Purpose	Official URL	Book Reference
Vortex Tracker II	AY music tracker	https://github.com/ivan-pirog/vortextracker	Chapter 11

See Also

- **Chapter 1:** First practical – setting up VS Code, sjasmplus, and the timing harness.
- **Chapter 22:** Agon Light 2 platform setup, eZ80 toolchain, Fab Agon Emulator.
- **Chapter 23:** DeZog integration with AI-assisted workflows.
- **Appendix A:** Z80 instruction reference – the instructions you will be debugging.
- **Appendix G:** AY register reference – useful when debugging sound code in ZEsarUX.

Appendix E: eZ80 Quick Reference

“The same instruction set, a completely different machine.” – Chapter 22

This appendix is a reference card for Z80 programmers approaching the eZ80 for the first time. It covers the architectural differences, the mode system, the new instructions, and the Agon Light 2 specifics you need for porting. It is not an exhaustive eZ80 manual — it is the subset that matters for the work in Chapter 22.

If you already know the Z80 (and if you have read this far, you do), the eZ80 will feel familiar. The registers have the same names, the instructions have the same mnemonics, the flags work the same way. But three things are different: addresses are 24 bits wide, registers can be 24 bits wide, and there is a mode system that controls which width is active. Everything else follows from that.

---

1. Architecture Overview

The eZ80 is Zilog’s extended Z80, designed for embedded systems that need more than 64 KB of address space. It is a strict superset of the Z80 — every Z80 opcode is valid on the eZ80, with identical behaviour. The extensions add 24-bit addressing, 24-bit register width, and a handful of new instructions.

Feature	Z80	eZ80
Address width	16-bit (64 KB)	24-bit (16 MB)
Register width	16-bit (HL, BC, DE, SP, IX, IY)	16-bit or 24-bit (mode-dependent)
Stack frame per PUSH/CALL	2 bytes	2 bytes (Z80 mode) or 3 bytes (ADL mode)
Hardware multiply	None	MLT rr (8x8 unsigned)
Instruction prefixes	CB, DD, ED, FD	Same, plus mode suffix prefixes
MBASE register	N/A	Provides upper 8 bits of addresses in Z80 mode
New instructions	N/A	LEA, PEA, MLT, TST, TSTIO, SLP, IN0/OUT0, STMIX/RSMIX

The key mental model: in **ADL mode** (Address Data Long), the eZ80 behaves like a Z80 with 24-bit registers and 24-bit addresses. In **Z80-compatible mode**, it behaves like a standard Z80, with 16-bit registers and the MBASE register providing the missing upper 8 address bits.

---

2. Mode System

The eZ80 has two operating modes that control register width and address generation. Understanding these modes is the single most important concept for any Z80 programmer approaching the eZ80.

The Two Modes

Z80-compatible mode. Registers are 16 bits wide. Addresses are 16 bits, with MBASE providing the upper 8 bits. `LD HL,$4000` loads a 16-bit value. `PUSH HL` pushes 2 bytes. Code behaves exactly like a standard Z80.

ADL mode (Address Data Long). Registers are 24 bits wide. Addresses are 24 bits. `LD HL,$040000` loads a 24-bit value. `PUSH HL` pushes 3 bytes. This is the native mode of the eZ80 and the default on the Agon Light 2.

Mode Suffixes

Individual instructions can override the current mode using suffix prefixes:

Suffix	Meaning	Register width	Address width
`.SIS`	Short Immediate, Short	16-bit	16-bit
`.LIS`	Long Immediate, Short	24-bit	16-bit
`.SIL`	Short Immediate, Long	16-bit	24-bit
`.LIL`	Long Immediate, Long	24-bit	24-bit

The first letter (S/L) controls register width for that instruction. The third letter (S/L) controls address width. In ADL mode, `.SIS` forces an instruction to behave as standard Z80. In Z80 mode, `.LIL` forces an instruction to behave as full 24-bit.

Mode-Switching Calls and Jumps

Calls and jumps can switch the processor mode at the target:

Instruction	Current mode	Target mode	Return address size
`CALL.IS nn`	ADL	Z80	3 bytes (ADL convention)
`CALL.IL nn`	Z80	ADL	3 bytes (long)
`JP.SIS nn`	any	Z80	N/A (no return address)
`JP.LIL nn`	any	ADL	N/A (no return address)
`RST.LIL $08`	Z80	ADL	3 bytes (long)

The `.IS` suffix means “Instruction Short” — the target runs in Z80 mode. `.IL` means “Instruction Long” — the target runs in ADL mode.

The Practical Rule

Stay in ADL mode. MOS boots the Agon in ADL mode. MOS API calls expect ADL mode. VDP commands are sent through MOS routines that assume 24-bit stack frames. If you switch to Z80 mode and call MOS, the stack frame width mismatch will corrupt the stack and crash.

If you need tight 16-bit loops (e.g., porting a Z80 inner loop without rewriting it), use the `.SIS` suffix on individual instructions rather than switching the entire processor mode.

The MBASE Trap

In Z80-compatible mode, the MBASE register provides the upper 8 bits of every memory address — including instruction fetches. If you change MBASE while executing in Z80 mode, the next instruction fetch uses the new MBASE value. Unless your code exists at the corresponding physical address, execution jumps into garbage.

Rule: if you must use Z80 mode, set MBASE once at startup and leave it alone. Better yet, stay in ADL mode.

---

3. New Instructions

These instructions exist on the eZ80 but not on the standard Z80. They are the reason the eZ80 is more than just a Z80 with wider registers.

Arithmetic and Test

Instruction	Bytes	Cycles	Description
`MLT BC`	2	6	8x8 unsigned multiply: B * C -> BC
`MLT DE`	2	6	8x8 unsigned multiply: D * E -> DE
`MLT HL`	2	6	8x8 unsigned multiply: H * L -> HL
`MLT SP`	2	6	8x8 unsigned multiply: SPH * SPL -> SP (rarely useful)
`TST A, n`	3	7	Test: A AND n, set flags, A unchanged
`TST A, r`	2	4	Test: A AND r, set flags, A unchanged
`TSTIO n`	3	12	Test I/O: (C) AND n, set flags

Address Computation

Instruction	Bytes	Cycles	Description
`LEA BC, IX+d`	3	4	BC = IX + signed displacement d
`LEA DE, IX+d`	3	4	DE = IX + d
`LEA HL, IX+d`	3	4	HL = IX + d
`LEA IX, IX+d`	3	4	IX = IX + d (add displacement to IX)
`LEA BC, IY+d`	3	4	BC = IY + d
`LEA DE, IY+d`	3	4	DE = IY + d
`LEA HL, IY+d`	3	4	HL = IY + d
`LEA IY, IY+d`	3	4	IY = IY + d (add displacement to IY)
`PEA IX+d`	3	7	Push (IX + d) onto stack
`PEA IY+d`	3	7	Push (IY + d) onto stack

LEA computes an effective address without performing a memory access — it is pure register arithmetic. On the standard Z80, computing `HL = IX + 5` requires a multi-instruction sequence (`PUSH IX / POP HL / LD DE,5 / ADD HL,DE`). LEA does it in a single instruction at 4 cycles.

I/O and System

Instruction	Bytes	Cycles	Description
`IN0 r, (n)`	3	7	Read from internal I/O port (8-bit address)
`OUT0 (n), r`	3	7	Write to internal I/O port (8-bit address)
`SLP`	2	–	Sleep: halt CPU until interrupt (lower power than HALT)
`STMIX`	2	4	Set mixed-mode flag (enable ADL/Z80 interleaving)
`RSMIX`	2	4	Reset mixed-mode flag

IN0/OUT0 use an 8-bit port address (unlike the standard IN/OUT which can use 16-bit port addresses via BC). They are designed for the eZ80's internal peripherals and are rarely used in Agon game code.

---

4. MLT — The Game Changer

Of all the eZ80's new instructions, MLT is the most impactful for game and demo code. It performs an 8x8 unsigned multiply in a single instruction.

On the standard Z80, 8x8 multiplication requires a shift-and-add loop:

```
; Z80: 8x8 unsigned multiply (B * C -> A:C)
; Cost: 196-204 T-states, 14 bytes
mulu_z80:
    ld   a, 0           ; 7T   clear accumulator
```

```
    ld   d, 8           ; 7T   8 bits

.loop:
    rr   c              ; 8T   shift multiplier bit into carry
    jr   nc, .noadd     ; 7/12T
    add  a, b           ; 4T   add multiplicand
.noadd:
    rra                 ; 4T   shift result right
    dec  d              ; 4T
    jr   nz, .loop      ; 12T
    ret                 ; 10T  ~200 T-states total
```

On the eZ80:

```
; eZ80: 8x8 unsigned multiply (B * C -> BC)
; Cost: 6 cycles, 2 bytes
    mlt  bc             ; BC = B * C. Done.
```

Six cycles. Two bytes. No loop, no carry management, no temporary registers. The result lands in the full 16-bit register pair: the high byte of the product is in B, the low byte in C.

What MLT Enables

Sine table indexing. Computing `base + angle * stride` for variable-stride sine lookups reduces from a subroutine call to two instructions (`MLT` + `ADD`).

Sprite offset calculation. Finding the address of sprite frame N in a sprite sheet: `base + frame * frame_size`. With MLT, this is trivial when frame_size fits in 8 bits.

Fixed-point arithmetic. Multiplying two 8-bit fixed-point values (e.g., velocity * friction) becomes a single instruction instead of a 200-T-state loop.

Tile map addressing. Computing `tile_base + (row * width + col)` where width fits in 8 bits: one MLT for the row offset, one ADD for the column.

MLT Limitations

- **Unsigned only.** For signed multiplication, adjust the sign manually after the MLT.
- **8x8 only.** For 16x16 multiply, you still need a multi-step algorithm (though you can build it from MLT components).
- **Result overwrites both operands.** `MLT BC` destroys both B and C, replacing them with the 16-bit product. Save the inputs first if you need them.

---

5. Key Differences Summary

Feature	Z80 (ZX Spectrum 128K)	eZ80 (Agon Light 2)
Clock	3.5 MHz	18.432 MHz
Address width	16-bit (64 KB visible)	24-bit (16 MB, 512 KB populated)

Feature	Z80 (ZX Spectrum 128K)	eZ80 (Agon Light 2)
Register width	16-bit	16-bit (Z80 mode) or 24-bit (ADL mode)
Stack per PUSH/CALL	2 bytes	2 bytes (Z80 mode) or 3 bytes (ADL mode)
Multiply instruction	None (shift-and-add loop, ~200T)	MLT rr (6 cycles)
RAM	128 KB banked (8 x 16 KB pages)	512 KB flat
RAM access model	Bank switching via port $7FFD	Flat 24-bit addressing
Video	ULA: direct memory-mapped at $4000	VDP (ESP32): command-based via UART
Sound	AY-3-8910 via I/O ports	VDP audio via serial commands
Interrupts	IM1 (RST $38) or IM2 (vector table)	IM2 with MOS-managed vectors
Frame budget (50 Hz)	~71,680 T-states (Pentagon)	~368,640 T-states
Contended memory	Yes (ULA steals cycles from $4000-$7FFF)	No contention
OS	None (bare metal)	MOS (Machine Operating System)
Storage	Tape / DivMMC	SD card (FAT32, MOS file API)

---

6. Agon Light 2 Specifics

Hardware

- **CPU:** Zilog eZ80F92, 18.432 MHz
- **RAM:** 512 KB external SRAM, flat address space
- **VDP:** ESP32-PICO-D4 running FabGL, communicates with eZ80 via UART at 1,152,000 baud
- **Video modes:** Multiple, up to 640x480x64 colours. Most games use 320x240 or 320x200.
- **Hardware sprites:** Up to 256, managed entirely by VDP
- **Hardware tilemaps:** Scrollable tile layers, managed by VDP
- **Audio:** VDP synthesis — sine, square, triangle, sawtooth, noise. Per-channel ADSR envelopes.
- **Storage:** MicroSD card, FAT32, accessed through MOS file API

Frame Budget

At 18.432 MHz and 50 Hz refresh:

```
18,432,000 cycles/sec / 50 frames/sec = 368,640 cycles/frame
```

Compare with Pentagon (3.5 MHz, 50 Hz): 71,680 T-states/frame. The Agon has roughly **5.1x** the per-frame budget. But since many eZ80 instructions also execute

in fewer cycles than their Z80 equivalents, effective throughput for typical code is 5x–20x greater.

MOS API

MOS (Machine Operating System) provides the system interface. The standard entry point is `RST $08`:

```
; MOS API call pattern (in ADL mode)
    ld   a, mos_function     ; function number in A
    rst  $08                 ; call MOS
    ; return value in A (and sometimes HL)
```

Key MOS API functions:

Function	Number	Description
`mos_getkey`	$00	Wait for keypress, return ASCII code
`mos_load`	$01	Load file from SD card to memory
`mos_save`	$02	Save memory region to file on SD card
`mos_cd`	$03	Change current directory
`mos_dir`	$04	List directory contents
`mos_del`	$05	Delete a file
`mos_ren`	$06	Rename a file
`mos_sysvars`	$08	Get pointer to system variables (keyboard map, VDP state, clock)
`mos_fopen`	$0A	Open file, return handle
`mos_fclose`	$0B	Close file handle
`mos_fread`	$0C	Read bytes from file
`mos_fwrite`	$0D	Write bytes to file
`mos_fseek`	$0E	Seek within file

VDP Command Protocol

VDP commands are sent as byte streams using `RST $10` (MOS: output byte to VDP). Most commands begin with VDU 23, followed by command-specific parameters:

```
; Send one byte to VDP
    ld   a, byte_value
    rst  $10                 ; MOS: send byte to VDP

; Example: set screen mode
    ld   a, 22               ; VDU 22 = set mode
    rst  $10
    ld   a, mode_number      ; 0-3 for standard modes
    rst  $10

; Example: move hardware sprite
; VDU 23, 27, 4, spriteNum          -- select sprite
```

```
; VDU 23, 27, 13, x.lo, x.hi, y.lo, y.hi  -- set position
```

The VDP processes commands asynchronously. There is a serial transfer delay between sending a command and the VDP acting on it. For smooth animation, send all updates early in the frame.

---

7. Porting Checklist

When porting Z80 code from the Spectrum to eZ80 ADL mode on the Agon, walk through this checklist:

Addresses and pointers: - All addresses become 24-bit (3 bytes instead of 2) - Change `DW` (define word) to `DL` (define long) for address tables - Pointer table indexing changes from `* 2` to `* 3` - Ensure upper byte of 24-bit addresses is correct (typically $00 or $04)

Stack frames: - Every PUSH is 3 bytes, every CALL pushes a 3-byte return address - Verify PUSH/POP pairs are balanced — a mismatched pair corrupts 3 bytes, not 2 - Stack-relative offsets (e.g., accessing parameters pushed by the caller) change

Block operations: - `LDIR` / `LDDR` use 24-bit BC in ADL mode — ensure the upper byte of BC is zero if your count fits in 16 bits - `PUSH`/`POP`-based block copy tricks push 3 bytes per PUSH, not 2

Multiply: - Replace shift-and-add multiply loops with `MLT` where applicable - `MLT BC` = B * C -> BC, `MLT DE` = D * E -> DE, `MLT HL` = H * L -> HL

I/O and peripherals: - Replace port I/O (`OUT (C), A`, `IN A, ($FE)`) with MOS/VDP API calls - Replace direct framebuffer writes ($4000–$5AFF) with VDP commands — Replace AY register writes (FFFD/$BFFD) with VDP sound commands

Memory architecture: - Remove all bank switching logic (port $7FFD writes) — flat address space - Remove contended memory workarounds — no contention on the Agon - Remove shadow screen tricks — VDP handles double buffering

Code patterns that become unnecessary: - Self-modifying code for speed (still works, rarely worth the complexity) - Stack pointer tricks for fast screen fills (no framebuffer to fill) - Pre-shifted sprite copies (hardware sprites handle sub-pixel positioning) - Interleaved screen address calculations (DOWN_HL, pixel_addr — delete them)

Code patterns that transfer directly: - Entity system loops (just widen pointers) - AABB collision detection (8-bit comparisons, unchanged) - Fixed-point 8.8 arithmetic (byte-level, unchanged) - State machines and jump tables (widen table entries to 24-bit) - DJNZ loops, CPIR searches, flag-based branching (all identical)

See Also

- **Appendix A: Z80 Instruction Quick Reference** — complete Z80 instruction table with T-states, byte counts, and flag effects. Everything in Appendix A also applies to the eZ80 in Z80-compatible mode.

- **Chapter 22: Porting — Agon Light 2** — the full porting walkthrough, with before/after code examples for rendering, sound, input, and game logic.

Sources: Zilog eZ80 CPU User Manual (UM0077); Zilog eZ80F92 Product Specification (PS0153); Agon Light 2 Official Documentation, The Byte Attic; Dean Belfield, "Agon Light — Programming Guide" (breakintoprogram.co.uk); Agon MOS API Documentation (github.com/AgonConsole8/agon-docs); Chapter 22 of this book

Appendix F: Z80 Variants — Extended Instruction Sets

"The same instruction set, a completely different machine." – Chapter 22

The Zilog Z80 did not stay frozen in 1976. Over five decades, the original design has been cloned, extended, reimagined, and — in the case of the ZX Spectrum Next — rebuilt by the very community that spent thirty years cursing its limitations. This appendix surveys the major Z80 variants and their instruction set extensions, with a focus on what matters to demoscene and game programmers. The standard Z80 instruction set is covered in Appendix A; the eZ80 gets a deep dive in Appendix E. This appendix is the big picture — how the variants relate, what each one adds, and why.

---

1. The Z80 Family Tree

The Z80 was designed by Federico Faggin and Masatoshi Shima at Zilog in 1976 as a software-compatible successor to the Intel 8080. It became the most widely used 8-bit CPU in history, powering everything from CP/M business machines to arcade cabinets to an entire generation of home computers. The instruction set was frozen at launch and never officially extended by Zilog — until the eZ80, two decades later.

But others did extend it. Here is the family at a glance:

Variant	Year	Notable Machine	Key Addition
Z80 (Zilog)	1976	ZX Spectrum, MSX, Amstrad CPC	The original. 158 documented instructions.
KR1858VM1 (Soviet)	~1986	Pentagon, Scorpion	Exact clone. No instruction changes.
NSC800 (National Semi)	1980	Various embedded	CMOS Z80 with 8085-style bus. No new instructions.
R800 (ASCII Corp)	1990	MSX turboR	Z80-compatible, radically different pipeline. MULUB, MULUW.

Variant	Year	Notable Machine	Key Addition
eZ80 (Zilog)	2001	Agon Light 2	24-bit addressing, MLT, LEA, PEA, ADL mode.
Z80N (Next team)	2017	ZX Spectrum Next	Demoscene wish list: MUL, MIRROR, LDIRX, PIXELDN, barrel shifts.

The Z80 and its clones share an identical instruction set. The R800, eZ80, and Z80N each added instructions to solve specific problems — but very different problems, reflecting very different design goals.

---

2. Z80N — The Demoscener's Wish List

The Z80N is the CPU in the ZX Spectrum Next. It was designed by Victor Trucco, Fabio Belavenuto, and the Next team — people who had spent decades writing Z80 code and knew exactly where it hurt. Every new instruction addresses a specific, documented pain point from thirty years of Spectrum programming. The Z80N runs at 28 MHz (8x the original Spectrum clock) and adds roughly 40 new instructions, all encoded in previously-unused `$ED xx` opcode space.

The best way to understand the Z80N extensions is by the problem each instruction solves.

Screen Navigation (the DOWN_HL problem)

On a standard Spectrum, calculating a screen address from pixel coordinates takes 50-60 T-states and a page of code (see `pixel_addr` in Appendix A). Moving one pixel row down requires the infamous `DOWN_HL` routine — a conditional maze of INC, AND, ADD, and SUB that handles character boundaries and third boundaries. The Z80N replaces all of this with single instructions.

Instruction	Bytes	T	What it replaces
`PIXELDN`	2	4	The 10+ instruction `DOWN_HL` sequence (check third boundary, handle wrap, adjust H and L). Moves HL one pixel row down in screen memory.
`PIXELAD`	2	4	Full screen address calculation from (D,E) coordinates. Replaces the `pixel_addr` routine (~55T, 15+ instructions).
`SETAE`	2	4	Sets the appropriate pixel bit in A based on the low 3 bits of E (the x-coordinate). Replaces a lookup table or shift sequence.

With these three instructions, the entire pixel-plotting sequence that consumed 70+ T-states and 20+ bytes on the original Z80 becomes:

```
; Z80N: plot pixel at (D=y, E=x) — 12T total
    pixelad             ; 4T  HL = screen address from (D,E)
    setae               ; 4T  A = pixel bit mask from E
    or   (hl)           ; 4T  set pixel (non-destructive)
    ; ... ld (hl), a to write
```

Sprite Rendering (the masked blit problem)

The single most CPU-intensive operation in any Spectrum game or demo is the masked sprite blit: copying a rectangular block of sprite data to screen memory while skipping transparent pixels. On the standard Z80, this requires an inner loop of LD/AND/OR/LD per byte, typically 30-40 T-states per pixel byte. The Z80N adds block-copy instructions with built-in transparency.

Instruction	Bytes	T	What it replaces
`LDIX`	2	5	`LDI` but skips the copy if `(HL) == A`. One-instruction transparent copy: load A with the transparent colour, point HL at source, DE at destination, and each byte is copied only if it is not the transparent value.
`LDDX`	2	5	Same as LDIX but decrementing (like `LDD`).
`LDIRX`	2	5/byte	Repeating LDIX. Hardware masked sprite blit in a single instruction. Copies BC bytes from (HL) to (DE), skipping any byte equal to A.
`LDDRX`	2	5/byte	Repeating LDDX.
`LDPIRX`	2	5/byte	Pattern fill with transparency from an 8-byte aligned source. Reads from `(HL & $FFF8) + (E & 7)`, copies to (DE) if not equal to A, increments DE, decrements BC. Hardware tiled background renderer.

`LDIRX` alone replaces the most heavily optimised inner loop in decades of Spectrum game code. `LDPIRX` is even more exotic — it treats the source as a repeating 8-byte pattern, effectively giving you a hardware tile renderer with transparency. Combined with the Next's Layer 2 and tilemap hardware, these instructions make the Spectrum Next a qualitatively different platform for sprite-heavy games.

Arithmetic (the multiply problem)

The Z80 has no multiply instruction. Every multiply in Spectrum code is a shift-and-add loop costing 150-250 T-states (see `mulu112` in Appendix A). The Z80N fixes

this with a single instruction.

Instruction	Bytes	T	What it replaces
`MUL D,E`	2	8	8x8 unsigned multiply, result in DE. Replaces the ~200T shift-and-add loop.

Eight T-states. At 28 MHz, that is 286 nanoseconds. The same operation on a standard 3.5 MHz Spectrum takes roughly 57 microseconds — a 200:1 improvement when you factor in both the faster clock and the faster instruction. Rotation matrices, coordinate transforms, texture mapping, perspective projection — everything that needs multiply is fundamentally cheaper on the Z80N.

Bit Manipulation

Instruction	Bytes	T	What it replaces
`MIRROR`	2	4	Reverses all 8 bits of A. Horizontal sprite flip without a 256-byte lookup table. On the standard Z80, bit-reversing A requires either an unrolled 18-instruction sequence (`LD B,A : XOR A` then 8× `RR B : RLA`, ~104T) or a 256-byte lookup table (11T but costs 256 bytes of RAM).
`SWAPNIB`	2	4	Swaps the high and low nibbles of A. Replaces `RLCA : RLCA : RLCA : RLCA` (16T, 4 bytes).
`TEST nn`	3	7	`AND A, nn` without storing the result — sets flags but preserves A. Like a CP for bitwise AND. Similar to the eZ80's TST instruction.

`MIRROR` is particularly valuable for games. Without it, every horizontally flipped sprite either needs a pre-flipped copy in memory (doubling sprite data) or a 256-byte bit-reversal lookup table plus per-byte table lookups. With it, you can flip sprites on the fly at 4T per byte.

Barrel Shifts (the multi-bit shift problem)

On the standard Z80, shifting a 16-bit value by more than one bit requires a loop: `SLA E : RL D` per bit, costing 16T per bit position. Shifting DE left by 5 bits costs 80T. The Z80N adds barrel shift instructions that shift DE by an arbitrary number of positions (specified in B) in constant time.

Instruction	Bytes	T	What it replaces
`BSLA DE,B`	2	4	Shift DE left by B bits. Replaces `B * (SLA E : RL D)`.
`BSRA DE,B`	2	4	Arithmetic shift DE right by B bits (sign-extending).
`BSRL DE,B`	2	4	Logical shift DE right by B bits (zero-filling).
`BSRF DE,B`	2	4	Shift DE right by B bits, filling with bit 15.
`BRLC DE,B`	2	4	Rotate DE left by B bits (circular).

These are enormously useful in fixed-point arithmetic, pixel sub-positioning, and any code that converts between integer scales. A common operation like "multiply by 5 and shift right 3" that would take ~50T on a standard Z80 becomes trivial.

Convenience Instructions

Instruction	Bytes	T	What it replaces
`PUSH nn`	4	11	Push a 16-bit immediate value onto the stack. No register needed. Saves the `LD rr, nn : PUSH rr` pattern (21T, 4 bytes on the original Z80 — same byte count, but 10T faster and does not clobber a register pair).
`ADD HL,A`	2	4	Add A to HL. Replaces the 5-instruction, 23T sequence: `ADD A,L : LD L,A : ADC A,H : SUB L : LD H,A` (or the equivalent using a spare register).
`ADD DE,A`	2	4	Same as ADD HL,A but for DE.
`ADD BC,A`	2	4	Same for BC.
`NEXTREG reg,val`	4	12	Direct write to a Next hardware register. No I/O port setup needed. Replaces `LD BC,$243B : OUT (C),reg : LD BC,$253B : OUT (C),val` — four instructions, 8 bytes, ~48T.
`NEXTREG reg,A`	3	8	Write A to a Next hardware register.
`OUTINB`	2	5	`OUT (C),(HL) : INC HL` combined. Useful for streaming data to I/O ports.

The Big Picture

The Z80N is, in a real sense, thirty years of demoscene frustration cast in silicon. Each instruction is a scar from a specific, well-documented pain:

- `PIXELDN` exists because every Spectrum programmer has written `DOWN_HL` at least once, debugged the third-boundary case at least twice, and wished they never had to again.
- `MIRROR` exists because every game programmer has wasted 256 bytes on a bit-reversal table for horizontal sprite flips.
- `LDIRX` exists because the masked blit inner loop is where most Spectrum games spend most of their CPU time.
- `MUL D,E` exists because the shift-and-add multiply loop is the single most re-implemented subroutine in Z80 history.

Unlike the eZ80 (designed by Zilog for embedded markets) or the R800 (designed by ASCII Corporation for the MSX platform), the Z80N was designed by the community, for the community. The instruction set reads like a demoscene wish list because it *is* a demoscene wish list — the Next team solicited input from active Spectrum coders and prioritised the instructions that would remove the most pain from the most common operations.

---

3. eZ80 — The Enterprise Extension

The eZ80 is Zilog's official successor to the Z80, designed for embedded systems that need more than 64 KB of address space. It is a strict superset of the Z80 — every Z80 opcode is valid and behaves identically. The extensions are architectural rather than computational:

- **24-bit addressing and ADL mode.** Registers can be 16-bit or 24-bit depending on the operating mode. ADL mode (Address Data Long) gives you 24-bit registers and a flat 16 MB address space. Z80-compatible mode behaves exactly like a standard Z80 with MBASE providing the missing upper 8 address bits.

- **MLT rr — 8x8 unsigned multiply.** `MLT BC` multiplies B by C and stores the 16-bit result in BC. Similarly for `MLT DE` (D * E -> DE) and `MLT HL` (H * L -> HL). This is more flexible than the Z80N's `MUL D,E`, which only operates on DE. The eZ80 gives you three independent multiply units. At 6 T-states on the eZ80 (running at 18.432 MHz on the Agon), this is blazing fast.

- **LEA and PEA.** Load Effective Address and Push Effective Address — indexed address computation instructions. `LEA rr, IX+d` loads the computed address into a register pair without accessing memory. `PEA IX+d` pushes the computed address onto the stack. Useful for parameter passing and pointer arithmetic.

- **TST (test) and TSTIO.** Non-destructive AND test, similar to the Z80N's `TEST nn`. `TSTIO` tests an I/O port value against a mask.

- **IN0/OUT0.** I/O access to the internal peripheral space (addresses 00 — —FF).

The eZ80 was designed for industrial control, networking equipment, and printers — not retrocomputing. But it landed in the Agon Light 2, and suddenly a CPU designed for embedded markets became a retro gaming platform. The full eZ80 reference is in Appendix E; the porting story is in Chapter 22.

---

4. R800 — The MSX turboR Speedster

The R800 is the oddest member of the family. Developed by ASCII Corporation for the MSX turboR (1990, Panasonic FS-A1GT/FS-A1ST), it is Z80-compatible in the sense that it executes the full Z80 instruction set — but its internal architecture is radically different.

Pipeline, not microcode. The original Z80 is microcoded: each instruction is broken into machine cycles (M-cycles) of 3-6 clock ticks each, and complex instructions take many M-cycles. The R800 uses a pipelined design where most instructions complete in 1-2 clock cycles. At 7.16 MHz, this gives it an effective throughput roughly 5-8x faster than a 3.5 MHz Z80 for typical code.

Hardware multiply. The R800 adds two multiply instructions:

Instruction	Operands	Result	Cycles	Notes
`MULUB A,r`	A * r (8-bit unsigned)	HL = 16-bit product	14	r = B, C, D, E
`MULUW HL,rr`	HL * rr (16-bit unsigned)	DE:HL = 32-bit product	36	rr = BC, SP

A 16x16 multiply with a 32-bit result in 36 cycles is remarkable for an 8-bit-era CPU. On a standard Z80, a 16x16 multiply takes 600-1000 T-states depending on implementation. The R800 makes 3D transformations, DSP-style filtering, and other multiply-heavy algorithms genuinely practical.

The pipeline trap. Code optimised for Z80 timing can behave unexpectedly on the R800. The Z80 optimisation trick of unrolling `LDIR` into individual `LDI` instructions (saving 5T per byte) actually runs *slower* on the R800, because the R800's pipeline handles the `LDIR` repeat prefix efficiently. Similarly, self-modifying code — a staple of Z80 demoscene technique — can stall the R800's pipeline when a write hits a prefetched instruction. Code that is fast on the Z80 is not necessarily fast on the R800, and vice versa.

Demoscene presence. The MSX turboR has a tiny but dedicated demoscene. The hardware multiply makes real-time 3D feasible, and the raw speed allows effects that would be impossible at Z80 clock rates. But the platform's rarity (only sold in Japan, small production run) means the R800 remains a footnote in the broader Z80 story.

---

5. Soviet Clones — Behind the Iron Curtain

The Soviet Union produced several Z80 clones to circumvent Western export restrictions. These chips enabled an entire ecosystem of ZX Spectrum-compatible computers that flourished from the late 1980s through the 1990s — and whose demoscene community remains active today.

KR1858VM1 (KP1858BM1). The primary Soviet Z80 clone. Pin-compatible, instruction-compatible, bug-compatible. Manufactured at the Angstrem factory

in Zelenograd using reverse-engineered masks. The KR1858VM1 powered the Pentagon 128 and Scorpion ZS-256 — the two most important Soviet Spectrum clones, and the platforms where much of the modern Russian/CIS ZX demoscene still targets its work.

T34VM1 (T34BM1). A later CMOS version of the same design, with lower power consumption and slightly different electrical characteristics. Functionally identical to the KR1858VM1.

Neither chip adds any new instructions. They are exact functional replicas of the Zilog Z80. The differences are electrical: different fabrication processes, different timing margins on setup and hold, slightly different behaviour on undocumented opcodes and flag bits. For software purposes, code that runs on a Zilog Z80 runs identically on a KR1858VM1.

The historical significance is immense. Without these clones, the ZX Spectrum ecosystem would not have spread across the Soviet Union and its successor states. The Pentagon 128, built around the KR1858VM1, became the *de facto* standard Spectrum platform in Russia, and its uncontended timing (no ULA contention delays) is the reference timing used throughout this book.

---

6. Comparison Table

Feature	Z80	Z80N	eZ80	R800
Clock (typical)	3.5 MHz	28 MHz	18.4 MHz	7.16 MHz
Address space	64 KB	64 KB + Next regs	16 MB	64 KB
Hardware multiply	No	`MUL D,E` (8T)	`MLT rr` (6T)	`MULUB` (14T), `MULUW` (36T)
16x16 multiply	No	No	No	`MULUW` (36T, 32-bit result)
Barrel shift	No	`BSLA/BSRA/BSRL/BSRF/BRLC DE,B` (4T)	No	No
Block copy with mask	No	`LDIRX`, `LDPIRX`	No	No
Screen address helpers	No	`PIXELDN`, `PIXELAD`, `SETAE`	No	No
Bit reverse	No	`MIRROR` (4T)	No	No
Nibble swap	No	`SWAPNIB` (4T)	No	No
24-bit mode	No	No	ADL mode	No
Push immediate	No	`PUSH nn` (11T)	No	No
Add 8-bit to 16-bit	No	`ADD HL/DE/BC, A` (4T)	No	No
Test (non-destructive AND)	No	`TEST nn` (7T)	`TST` (7T)	No
Hardware register I/O	No	`NEXTREG` (8-12T)	`IN0`/`OUT0`	No

Feature	Z80	Z80N	eZ80	R800
Designed for	General computing	ZX Spectrum Next	Embedded systems	MSX turboR

Effective Multiply Performance

Since all four variants run at different clock speeds, the raw T-state counts do not tell the full story. Here is the wall-clock time for an 8x8 unsigned multiply on each platform:

Variant	Clock	Instruction	Cycles	Wall-clock time
Z80	3.5 MHz	Shift-and-add loop	~200	~57 us
Z80N	28 MHz	`MUL D,E`	8	~0.29 us
eZ80	18.4 MHz	`MLT DE`	6	~0.33 us
R800	7.16 MHz	`MULUB A,r`	14	~1.96 us

The Z80N and eZ80 are effectively tied for multiply performance. The R800 is 30x faster than a standard Z80 but 6-7x slower than the Z80N/eZ80. All three are fast enough to make real-time 3D practical.

---

7. What This Means for the Book

All assembly code in this book targets the **standard Z80 instruction set**. Every example in every chapter will assemble and run on a stock ZX Spectrum 48K, a Pentagon 128, a Scorpion, an MSX, or any other machine with a Zilog Z80 or compatible clone. No extensions required.

This is a deliberate choice. The optimisation principles — T-state budgets, register allocation, loop structure, self-modifying code, unrolled loops, stack tricks — are universal. Code that is fast on a 3.5 MHz Z80 is fast on a 28 MHz Z80N. Code that fits in 48 KB fits in 16 MB. The constraints of the original Z80 teach you to think in a way that transfers to every variant in the family.

That said, if you have a ZX Spectrum Next, the Z80N extensions are too good to ignore. Chapter 22 covers porting strategies including Z80N-specific optimisations. If you have an Agon Light 2, Appendix E is your eZ80 reference and Chapter 22 walks through a complete Spectrum-to-Agon port. The barrel shifts, hardware multiply, and masked blit instructions do not change *how* you think about optimisation — they change *where the bottleneck moves* once the classic pain points are eliminated.

The fundamentals do not change. The instructions get better.

See Also

- **Appendix A: Z80 Instruction Quick Reference** — complete standard Z80 instruction table with T-states, byte counts, and flag effects. The baseline that

all variants share.

- **Appendix E: eZ80 Quick Reference** — full eZ80 reference including mode system, MLT, LEA/PEA, and Agon Light 2 specifics.
- **Chapter 22: Porting — Agon Light 2** — practical porting walkthrough covering both eZ80 and Z80N extensions in context.

Sources: Zilog Z80 CPU User Manual (UM0080); Zilog eZ80 CPU User Manual (UM0077); ZX Spectrum Next User Manual, Issue 2; ZX Spectrum Next Extended Instruction Set Documentation (wiki.specnext.dev); Victor Trucco, Fabio Belavenuto et al., Z80N instruction set design notes; ASCII Corporation R800 Technical Reference (1990); Sean Young, "The Undocumented Z80 Documented" (2005); Introspec, "Once more about DOWN_HL" (Hype, 2020); Dark / X-Trade, "Programming Algorithms" (Spectrum Expert #01, 1997)

Appendix G: AY-3-8910 / TurboSound / Triple AY Register Reference

> *"Fourteen registers. Three square-wave tone channels. One noise generator. One envelope generator. That is all you get."* – Chapter 11

This appendix is a complete register-level reference for the AY-3-8910 Programmable Sound Generator as used in the ZX Spectrum 128K, TurboSound-equipped clones (Pentagon, Scorpion), and the ZX Spectrum Next's triple-AY configuration. Chapter 11 covers the musical techniques; this appendix is the datasheet you keep open while coding.

All hex values use the `$FF` notation. Binary values use `%10101010`. Clock frequency assumes the PAL Spectrum 128K standard of 1.7734 MHz unless otherwise noted.

I/O Ports on ZX Spectrum 128K

Port	Direction	Function
$FFFD$ $*$ $*$|$Write$|$Selectactiveregister$(0—15)||$ $*$ $*$**FFFD**	Read	Read value of currently selected register
$BFFD	Write	Write data to selected register

The Write Sequence

Every AY register write is a two-step operation: select the register, then write the value.

```
; Write value E to AY register A
; Clobbers: BC
ay_write:
    ld   bc, $FFFD
    out  (c), a       ; 12T  select register
    ld   b, $BF       ; 7T   BC = $BFFD (C stays $FD)
```

```
    out  (c), e       ; 12T  write data
    ret               ;      total: 31T + call overhead
```

The trick: only the high byte of BC changes between the two `OUT` instructions. The low byte `$FD` stays in C throughout. This saves 3 bytes and 10 T-states compared to loading a full 16-bit value twice.

Reading a Register

```
; Read AY register A into A
; Clobbers: BC
ay_read:
    ld   bc, $FFFD
    out  (c), a       ; select register
    in   a, (c)       ; read value
    ret
```

Reading is useful for preserving the mixer state (R7) when modifying individual channel enables.

Bulk Register Write

For music engines that update all 14 registers per frame, an unrolled loop is fastest:

```
; Write all 14 AY registers from a buffer
; HL = pointer to 14-byte register buffer (R0-R13)
ay_flush:
    ld   de, $BFFD         ; D = $BF, E = $FD
    ld   c, e              ; C = $FD (shared low byte)
    ld   b, $FF             ; BC = $FFFD (register select port)
    xor  a                  ; start at R0
.loop:
    out  (c), a             ; select register A
    ld   b, d               ; BC = $BFFD
    outi                    ; write (HL) to port, inc HL, dec B
    ; B is now $BE after OUTI dec, but we reload it anyway
    ld   b, $FF             ; BC = $FFFD
    inc  a
    cp   14
    jr   nz, .loop
    ret
```

Complete Register Map

Overview

Reg	Name	Bits Used	R/W	Reset	Description
R0	Tone A period, low	7-0	R/W	$00	Channel A tone period, bits 7-0

Reg	Name	Bits Used	R/W	Reset	Description
R1	Tone A period, high	3-0	R/W	$00	Channel A tone period, bits 11-8
R2	Tone B period, low	7-0	R/W	$00	Channel B tone period, bits 7-0
R3	Tone B period, high	3-0	R/W	$00	Channel B tone period, bits 11-8
R4	Tone C period, low	7-0	R/W	$00	Channel C tone period, bits 7-0
R5	Tone C period, high	3-0	R/W	$00	Channel C tone period, bits 11-8
R6	Noise period	4-0	R/W	$00	Noise generator period (0-31)
R7	Mixer / I/O	7-0	R/W	$FF	Tone/noise enable + I/O port direction
R8	Volume A	4-0	R/W	$00	Channel A volume or envelope mode
R9	Volume B	4-0	R/W	$00	Channel B volume or envelope mode
R10	Volume C	4-0	R/W	$00	Channel C volume or envelope mode
R11	Envelope period, low	7-0	R/W	$00	Envelope period, bits 7-0
R12	Envelope period, high	7-0	R/W	$00	Envelope period, bits 15-8
R13	Envelope shape	3-0	W	–	Envelope waveform (writing restarts envelope)
R14	I/O Port A	7-0	R/W	–	General-purpose I/O (directly mapped on AY-3-8910)
R15	I/O Port B	7-0	R/W	–	General-purpose I/O (directly mapped on AY-3-8910)

Note on R14-R15: The AY-3-8910 has two 8-bit I/O ports. On the ZX Spectrum 128K, I/O port A (R14) is active – it reads the keyboard matrix and other hardware signals. R15 (I/O port B) is typically not connected. The AY-3-8912 (used in some clones) has only port A; the AY-3-8913 has no I/O ports at all. For sound

programming, R14 and R15 are irrelevant.

R0-R1: Channel A Tone Period (12-bit)

```
R0:  [D7] [D6] [D5] [D4] [D3] [D2] [D1] [D0]   bits 7-0 of period
R1:  [ 0] [ 0] [ 0] [ 0] [D11][D10][ D9][ D8]   bits 11-8 of period
```

- **Range:** 1 to 4095 (12-bit unsigned). Period of 0 behaves as 1 on most implementations.
- **Effect:** Sets the frequency of the square wave on channel A.
- **Formula:** `frequency = 1773400 / (16 * period)` (PAL Spectrum 128K)
- **Practical range:** Period 1 = 110,837 Hz (ultrasonic), period 4095 = 27 Hz (deep bass).

```
; Set channel A to middle C (C4, ~262 Hz, period 424)
    ld   a, 0              ; R0 = Tone A low
    ld   e, $A8            ; 424 & $FF = $A8
    call ay_write
    ld   a, 1              ; R1 = Tone A high
    ld   e, $01            ; 424 >> 8 = $01
    call ay_write
```

R2-R3: Channel B Tone Period (12-bit)

Identical layout to R0-R1, for channel B.

R4-R5: Channel C Tone Period (12-bit)

Identical layout to R0-R1, for channel C.

R6: Noise Period (5-bit)

```
R6:  [ 0] [ 0] [ 0] [D4] [D3] [D2] [D1] [D0]
```

- **Range:** 0 to 31. A single noise generator shared by all three channels.
- **Effect:** Lower values = higher-pitched noise (hiss). Higher values = lower-pitched, rougher noise.
- **Formula:** `noise_frequency = 1773400 / (16 * period)` (same as tone formula)

R6 Value	Character	Typical Use
0-5	High hiss	Hi-hat, cymbal, metallic ring
6-12	Mid noise	Snare drum body, white noise
13-20	Low rumble	Explosion, engine, wind
21-31	Very low	Thunder, distant rumble

The noise generator uses a 17-bit linear feedback shift register (LFSR) to produce pseudo-random output. The output is the same for both the AY-3-8910 and YM2149, but the LFSR tap positions differ, producing subtly different noise textures on each chip.

R7: Mixer Control (The Most Important Register)

```
R7:  [IOB] [IOA] [NC] [NB] [NA] [TC] [TB] [TA]
      bit7  bit6  bit5 bit4 bit3 bit2 bit1 bit0
```

Bit	Name	0 =	1 =
7	I/O port B direction	Output	**Input**
6	I/O port A direction	Output	**Input**
5	Noise C enable	**ON**	Off
4	Noise B enable	**ON**	Off
3	Noise A enable	**ON**	Off
2	Tone C enable	**ON**	Off
1	Tone B enable	**ON**	Off
0	Tone A enable	**ON**	Off

Critical: 0 means ON. The mixer uses active-low logic. A cleared bit enables the corresponding source. This is the single most confusing aspect of AY programming.

Bits 7-6: Always set to 1 (input mode) on the Spectrum. Do not change these unless you know what you are doing with the I/O ports.

Common Mixer Values

Value	Binary (`NC NB NA TC TB TA`)	Effect
`$38`	`%00 111 000`	All three tones, no noise
`$3E`	`%00 111 110`	Tone A only
`$3D`	`%00 111 101`	Tone B only
`$3B`	`%00 111 011`	Tone C only
`$36`	`%00 110 110`	Tone A + Noise A
`$2D`	`%00 101 101`	Tone B + Noise B
`$1B`	`%00 011 011`	Tone C + Noise C
`$28`	`%00 101 000`	All tones + Noise C (music + drums on C)
`$07`	`%00 000 111`	All noise, no tones
`$00`	`%00 000 000`	Everything on
`$3F`	`%00 111 111`	Everything off (silence)

Note: The binary values above show bits 5-0 only. Bits 7-6 should be `%11` for normal operation (I/O ports as inputs), making e.g. `$38` actually `%11 111 000` = `$F8`. However, on the Spectrum 128K the unused upper bits are ignored by convention, and most code uses the short form. Some engines write the full `$F8` form; both work identically.

```
; Enable tone A, tone B, tone C, and noise on C only
; Binary: I/O=11, NC=0(on), NB=1, NA=1, TC=0(on), TB=0(on), TA=0(on)
; = %11 011 000 = $D8 (full form) or $28 (short form, bits 7-6 treated as 0)
    ld   a, 7              ; R7 = Mixer
    ld   e, $28
    call ay_write
```

R8-R10: Volume Registers (5-bit)

```
R8:  [ 0] [ 0] [ 0] [ M] [D3] [D2] [D1] [D0]   Channel A
R9:  [ 0] [ 0] [ 0] [ M] [D3] [D2] [D1] [D0]   Channel B
R10: [ 0] [ 0] [ 0] [ M] [D3] [D2] [D1] [D0]   Channel C
```

Bit	Function
4 (M)	Mode: 0 = use fixed volume (bits 3-0), 1 = use envelope generator
3-0	Fixed volume level: 0 (silent) to 15 (maximum)

- **Fixed volume mode** (bit 4 = 0): Bits 3-0 set the channel volume directly. 15 = loudest, 0 = silent. The volume curve is logarithmic on a genuine AY-3-8910 (approximately 1.5 dB per step) but linear on the YM2149.
- **Envelope mode** (bit 4 = 1): The channel's volume is controlled by the envelope generator (R11-R13). Bits 3-0 are ignored. Only one envelope generator exists, shared by all channels in envelope mode.

Value	Effect
`$00`	Silent
`$08`	Half volume (approximately)
`$0F`	Maximum fixed volume
`$10`	Envelope mode (any value $10-$1F activates envelope)

4-bit DAC trick: By rapidly writing successive sample values to a volume register (without envelope mode), you can play digitised audio through the AY. At 8 kHz sample rate, this consumes approximately 437 T-states per sample write, leaving almost no CPU for other work. See Chapter 12 for the hybrid drum technique.

```
; Set channel A to maximum fixed volume
    ld   a, 8              ; R8 = Volume A
    ld   e, $0F            ; volume 15
    call ay_write

; Switch channel C to envelope mode
    ld   a, 10             ; R10 = Volume C
    ld   e, $10            ; bit 4 set = envelope mode
    call ay_write
```

R11-R12: Envelope Period (16-bit)

```
R11: [D7] [D6] [D5] [D4] [D3] [D2] [D1] [D0]   bits 7-0
R12: [D7] [D6] [D5] [D4] [D3] [D2] [D1] [D0]   bits 15-8
```

- **Range:** 1 to 65535 (16-bit unsigned). Period of 0 behaves as 1.
- **Formula:** `envelope_frequency = 1773400 / (256 * period)`

- **At period = 1:** ~6,927 Hz (one complete envelope cycle in ~144 microseconds)
- **At period = 65535:** ~0.106 Hz (one cycle in ~9.4 seconds)

The envelope period controls how fast the volume ramps up or down according to the shape selected in R13. For buzz-bass, the envelope period replaces the tone period as the pitch control:

```
bass_envelope_period = 1773400 / (256 * desired_frequency)
```

For bass C2 (65.4 Hz): period = 1773400 / (256 * 65.4) = 106.

```
; Set envelope period to 106 (bass C2 for buzz-bass)
    ld   a, 11             ; R11 = Envelope period low
    ld   e, 106            ; $6A
    call ay_write
    ld   a, 12             ; R12 = Envelope period high
    ld   e, 0
    call ay_write
```

R13: Envelope Shape (4-bit, Write-Only)

```
R13: [ 0] [ 0] [ 0] [ 0] [CONT] [ATT] [ALT] [HOLD]
                           bit3   bit2  bit1   bit0
```

Bit	Name	Function
3	CONT	Continue: 0 = single-shot (reset to 0 or hold), 1 = repeating
2	ATT	Attack: 0 = start at max, ramp down, 1 = start at 0, ramp up
1	ALT	Alternate: 0 = same direction each cycle, 1 = reverse direction
0	HOLD	Hold: 0 = cycle normally, 1 = hold at final level after first cycle

Critical behaviour: Writing ANY value to R13 immediately restarts the envelope from the beginning of the cycle. This is true even if you write the same value already stored. This restart behaviour is essential for triggering buzz-bass notes and drum sounds.

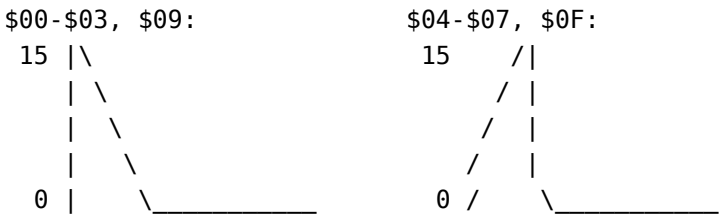
All 16 Shape Values

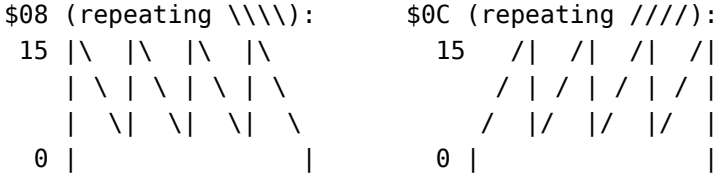
Although 4 bits allow 16 values, many produce identical output. There are only 10 unique shapes:

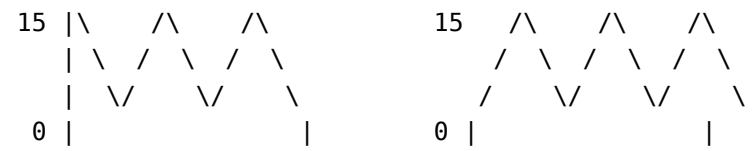
Value	Bits (CONT ATT ALT HOLD)	Shape	Behaviour
`$00`	`0 0 0 0`	`\___`	Decay to 0, hold at 0
`$01`	`0 0 0 1`	`\___`	Same as $00
`$02`	`0 0 1 0`	`\___`	Same as $00

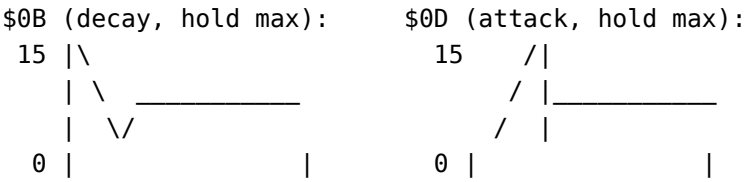
Value	Bits (CONT ATT ALT HOLD)	Shape	Behaviour
$03	0 0 1 1	___	Same as $00
$04	0 1 0 0	/___	Attack to 15, drop to 0, hold at 0
$05	0 1 0 1	/___	Same as $04
$06	0 1 1 0	/___	Same as $04
$07	0 1 1 1	/___	Same as $04
$08	1 0 0 0	\\\\	Repeating sawtooth down
$09	1 0 0 1	___	Single decay, hold at 0 (same as $00)
$0A	1 0 1 0	\/\/	Repeating triangle (down-up)
$0B	1 0 1 1	\^^^	Single decay, then hold at max (15)
$0C	1 1 0 0	////	Repeating sawtooth up
$0D	1 1 0 1	/^^^	Single attack, then hold at max (15)
$0E	1 1 1 0	/\/\	Repeating triangle (up-down)
$0F	1 1 1 1	/___	Single attack, drop to 0, hold at 0 (same as $04)

Envelope Shape Waveforms









Practical Envelope Uses

Shape	Value	Use Case
`$00`	`\___`	Drum decay: sharp hit that fades to silence
`$08`	`\\\\`	Buzz-bass: thick repeating bass tone
`$0C`	`////`	Buzz-bass (inverted phase): same pitch, different timbre
`$0A`	`\/\/`	Metallic/bell tone (triangle modulation)
`$0E`	`/\/\`	Metallic/bell tone (inverted triangle)
`$0D`	`/^^^`	Fade-in: volume rises to max and holds

Tone Period to Frequency Conversion

Formula

```
frequency = AY_clock / (16 * period)
period    = AY_clock / (16 * frequency)
```

AY Clock by Platform

Platform	AY Clock	CPU Clock	Relationship
ZX Spectrum 128K (PAL)	1,773,400 Hz	3,546,900 Hz	AY = CPU / 2
ZX Spectrum 128K (NTSC)	1,789,772 Hz	3,579,545 Hz	AY = CPU / 2
Pentagon 128	1,750,000 Hz	3,500,000 Hz	AY = CPU / 2
Amstrad CPC	1,000,000 Hz	4,000,000 Hz	AY = CPU / 4
MSX	1,789,772 Hz	3,579,545 Hz	AY = CPU / 2
Atari ST (YM2149)	2,000,000 Hz	8,000,000 Hz	YM = CPU / 4
ZX Spectrum Next	1,773,400 Hz	varies	Same as 128K

Practical consequence: The same period value produces slightly different pitches on different platforms. A tune composed on Pentagon (1.75 MHz) will sound fractionally sharp on a Spectrum 128K (1.7734 MHz). For most musical purposes, the difference is inaudible.

Complete Note Frequency Table

All period values calculated for AY clock = 1,773,400 Hz (PAL Spectrum 128K).

Octaves 1-2 cover the bass range (buzz-bass territory). Octaves 3-6 are the main melodic range. Octaves 7-8 are high-pitched and increasingly inaccurate due to integer rounding.

Octave 1 (Deep Bass)

Note	Freq (Hz)	Period	R_LO	R_HI
C-1	32.70	3390	$3E	$0D
C#1	34.65	3199	$7F	$0C
D-1	36.71	3019	$CB	$0B
D#1	38.89	2850	$22	$0B
E-1	41.20	2690	$82	$0A
F-1	43.65	2539	$EB	$09
F#1	46.25	2396	$5C	$09
G-1	49.00	2262	$D6	$08
G#1	51.91	2135	$57	$08
A-1	55.00	2015	$DF	$07
A#1	58.27	1902	$6E	$07
B-1	61.74	1795	$03	$07

Octave 2 (Bass)

Note	Freq (Hz)	Period	R_LO	R_HI
C-2	65.41	1695	$9F	$06
C#2	69.30	1599	$3F	$06
D-2	73.42	1510	$E6	$05
D#2	77.78	1425	$91	$05
E-2	82.41	1345	$41	$05
F-2	87.31	1270	$F6	$04
F#2	92.50	1198	$AE	$04
G-2	98.00	1131	$6B	$04
G#2	103.83	1067	$2B	$04
A-2	110.00	1008	$F0	$03
A#2	116.54	951	$B7	$03
B-2	123.47	897	$81	$03

Octave 3

Note	Freq (Hz)	Period	R_LO	R_HI
C-3	130.81	847	$4F	$03
C#3	138.59	800	$20	$03
D-3	146.83	755	$F3	$02
D#3	155.56	713	$C9	$02
E-3	164.81	673	$A1	$02
F-3	174.61	635	$7B	$02
F#3	185.00	599	$57	$02
G-3	196.00	566	$36	$02
G#3	207.65	534	$16	$02
A-3	220.00	504	$F8	$01
A#3	233.08	475	$DB	$01
B-3	246.94	449	$C1	$01

Octave 4 (Middle Octave)

Note	Freq (Hz)	Period	R_LO	R_HI
C-4	261.63	424	$A8	$01
C#4	277.18	400	$90	$01
D-4	293.66	378	$7A	$01
D#4	311.13	357	$65	$01
E-4	329.63	337	$51	$01
F-4	349.23	318	$3E	$01
F#4	369.99	300	$2C	$01
G-4	392.00	283	$1B	$01
G#4	415.30	267	$0B	$01
A-4	440.00	252	$FC	$00
A#4	466.16	238	$EE	$00
B-4	493.88	225	$E1	$00

Octave 5

Note	Freq (Hz)	Period	R_LO	R_HI
C-5	523.25	212	$D4	$00
C#5	554.37	200	$C8	$00
D-5	587.33	189	$BD	$00
D#5	622.25	178	$B2	$00
E-5	659.26	168	$A8	$00
F-5	698.46	159	$9F	$00
F#5	739.99	150	$96	$00
G-5	783.99	142	$8E	$00
G#5	830.61	134	$86	$00
A-5	880.00	126	$7E	$00
A#5	932.33	119	$77	$00
B-5	987.77	112	$70	$00

Octave 6

Note	Freq (Hz)	Period	R_LO	R_HI
C-6	1046.50	106	$6A	$00
C#6	1108.73	100	$64	$00
D-6	1174.66	94	$5E	$00
D#6	1244.51	89	$59	$00
E-6	1318.51	84	$54	$00
F-6	1396.91	79	$4F	$00
F#6	1479.98	75	$4B	$00
G-6	1567.98	71	$47	$00
G#6	1661.22	67	$43	$00
A-6	1760.00	63	$3F	$00
A#6	1864.66	59	$3B	$00
B-6	1975.53	56	$38	$00

Octave 7 (High)

Note	Freq (Hz)	Period	R_LO	R_HI
C-7	2093.00	53	$35	$00
C#7	2217.46	50	$32	$00
D-7	2349.32	47	$2F	$00
D#7	2489.02	45	$2D	$00
E-7	2637.02	42	$2A	$00
F-7	2793.83	40	$28	$00
F#7	2959.96	37	$25	$00
G-7	3135.96	35	$23	$00
G#7	3322.44	33	$21	$00
A-7	3520.00	31	$1F	$00
A#7	3729.31	30	$1E	$00
B-7	3951.07	28	$1C	$00

Octave 8 (Very High – Limited Accuracy)

Note	Freq (Hz)	Period	R_LO	R_HI	Actual Freq	Error (cents)
C-8	4186.01	26	$1A	$00	4264.23	+32
C#8	4434.92	25	$19	$00	4433.50	-1
D-8	4698.63	24	$18	$00	4618.23	-30
D#8	4978.03	22	$16	$00	5038.07	+21
E-8	5274.04	21	$15	$00	5278.27	+1
F-8	5587.65	20	$14	$00	5541.88	-14
F#8	5919.91	19	$13	$00	5833.55	-26
G-8	6271.93	18	$12	$00	6158.75	-32

Note: Above octave 6, integer rounding of the period value produces increasingly audible pitch errors. The “Actual Freq” and “Error” columns for octave 8 show the real output frequency and deviation in cents. For high notes, fine-tuning is not possible – the resolution is simply too coarse.

Compact Note Table for Z80 Code

Most tracker engines store the period table as 16-bit words. Here is a practical 96-note table (octaves 1-8) for inclusion in assembly source:

```
; AY note period table: 96 notes, C-1 to B-8
; Each entry is a 16-bit period value (low byte first)
; Index = (octave * 12) + semitone, where C=0, C#=1, ..., B=11
note_table:
    ; Octave 1
    DW 3390, 3199, 3019, 2850, 2690, 2539
    DW 2396, 2262, 2135, 2015, 1902, 1795
    ; Octave 2
    DW 1695, 1599, 1510, 1425, 1345, 1270
    DW 1198, 1131, 1067, 1008,  951,  897
    ; Octave 3
    DW  847,  800,  755,  713,  673,  635
    DW  599,  566,  534,  504,  475,  449
    ; Octave 4
    DW  424,  400,  378,  357,  337,  318
    DW  300,  283,  267,  252,  238,  225
    ; Octave 5
    DW  212,  200,  189,  178,  168,  159
    DW  150,  142,  134,  126,  119,  112
    ; Octave 6
    DW  106,  100,   94,   89,   84,   79
    DW   75,   71,   67,   63,   59,   56
    ; Octave 7
    DW   53,   50,   47,   45,   42,   40
    DW   37,   35,   33,   31,   30,   28
    ; Octave 8
    DW   26,   25,   24,   22,   21,   20
    DW   19,   18,   17,   16,   15,   14

; Look up note period: A = note number (0-95)
; Returns DE = period value
; Clobbers: HL
get_note_period:
    ld   h, 0
    ld   l, a
    add  hl, hl           ; HL = note * 2 (word index)
    ld   de, note_table
    add  hl, de
    ld   e, (hl)
    inc  hl
    ld   d, (hl)          ; DE = period
    ret
```

Table #5: Natural Tuning (Just Intonation)

The standard note table above uses equal temperament (12-TET) – every semitone is the 12th root of 2 apart. This works well for tone channels, but creates a problem for **buzz-bass (T+E)**: since `envelope_period = tone_period / 16`, any tone period

not divisible by 16 introduces a rounding error. The envelope drifts against the tone, producing audible beating on sustained bass notes.

Ivan Roshin's "Frequency Table with Zero Error" (2001) and oisee's VTi implementation (2009) solve this by using **just intonation** – integer-ratio intervals for C major / A minor:

```
C [9/8] D [10/9] E [16/15] F [9/8] G [10/9] A [9/8] B [16/15] C
```

This produces pure fifths (C–G, E–B, A–E at exact 3:2) and, critically, periods where most main-scale notes divide evenly by 16.

AY clock: 1,520,640 Hz (non-standard; select per-key frequency below):

```
; Table #5: Natural tuning note table
; 96 notes, C-1 to B-8, for AY clock = 1,520,640 Hz
; C major / A minor only; other keys via chip frequency change
natural_note_table:
    ; Octave 1
    DW 2880, 2700, 2560, 2400, 2304, 2160
    DW 2025, 1920, 1800, 1728, 1620, 1536
    ; Octave 2
    DW 1440, 1350, 1280, 1200, 1152, 1080
    DW 1013,  960,  900,  864,  810,  768
    ; Octave 3
    DW  720,  675,  640,  600,  576,  540
    DW  506,  480,  450,  432,  405,  384
    ; Octave 4
    DW  360,  338,  320,  300,  288,  270
    DW  253,  240,  225,  216,  203,  192
    ; Octave 5
    DW  180,  169,  160,  150,  144,  135
    DW  127,  120,  113,  108,  101,   96
    ; Octave 6
    DW   90,   84,   80,   75,   72,   68
    DW   63,   60,   56,   54,   51,   48
    ; Octave 7
    DW   45,   42,   40,   38,   36,   34
    DW   32,   30,   28,   27,   25,   24
    ; Octave 8
    DW   23,   21,   20,   19,   18,   17
    DW   16,   15,   14,   14,   13,   12
```

Period divisibility check (Octave 2, bass range):

Note	Period	mod 16	Env Period	Clean T+E?
C2	1440	0	90	Yes
C#2	1350	6	84	No
D2	1280	0	80	Yes
D#2	1200	0	75	Yes
E2	1152	0	72	Yes
F2	1080	0	67	~
F#2	1013	5	63	No
G2	960	0	60	Yes

Note	Period	mod 16	Env Period	Clean T+E?
G#2	900	4	56	No
A2	864	0	54	Yes
A#2	810	2	50	No
B2	768	0	48	Yes

Seven of twelve notes (all natural notes of C major) divide cleanly – compared to *zero* in the equal-tempered table.

Transposition via chip frequency: since the table is fixed to C/Am, other keys require a different AY clock. Each step multiplies by 2^(1/12):

Key	Chip Frequency (Hz)
C/Am	1,520,640
C#/A#m	1,611,062
D/Bm	1,706,861
D#/Cm	1,808,356
E/C#m	1,915,886
F/Dm	2,029,811
F#/D#m	2,150,510
G/Em	2,278,386
G#/Fm	2,413,866
A/F#m	2,557,401
A#/Gm	2,709,472
B/G#m	2,870,586

On hardware the AY clock is fixed; in trackers (Vortex Tracker II/Improved) and emulators, this is a per-module setting. Table #5 is the default for the autosiril MIDI-to-PT3 converter because most converted tracks use buzz-bass heavily. See Chapter 11 for a detailed explanation of the T+E alignment problem.

TurboSound: 2 x AY

TurboSound is a hardware modification adding a second AY chip, giving 6 tone channels, 2 noise generators, and 2 envelope generators. The most common modern implementation is the NedoPC TurboSound card.

Chip Selection

Both AY chips share the same I/O ports ($FFFD$/BFFD). A chip select value written to $FFFD switches all subsequent register operations to the selected chip.

Chip Select Value	Target
`$FF`	Chip 0 (primary / original AY)
`$FE`	Chip 1 (secondary AY)

```
; Select chip 0 (primary)
    ld   bc, $FFFD
    ld   a, $FF
    out  (c), a          ; all subsequent R/W goes to chip 0

; Select chip 1 (secondary)
    ld   bc, $FFFD
    ld   a, $FE
    out  (c), a          ; all subsequent R/W goes to chip 1
```

Important: The chip select persists until changed. After selecting chip 1, all register operations – including reads – target chip 1. Always explicitly select the chip before any register access in your engine.

TurboSound Engine Architecture

A typical TurboSound music engine maintains two 14-byte register buffers and writes them sequentially:

```
; Update both AY chips -- called once per frame from ISR
ts_update:
    ; --- Chip 0 ---
    ld   a, $FF
    ld   bc, $FFFD
    out  (c), a              ; select chip 0
    ld   hl, ay0_regs        ; 14-byte buffer for chip 0
    call ay_flush             ; write all 14 registers

    ; --- Chip 1 ---
    ld   a, $FE
    ld   bc, $FFFD
    out  (c), a              ; select chip 1
    ld   hl, ay1_regs        ; 14-byte buffer for chip 1
    call ay_flush             ; write all 14 registers
    ret

ay0_regs: DS 14              ; chip 0 register shadow
ay1_regs: DS 14              ; chip 1 register shadow
```

Total cost: approximately 28 OUT instructions, roughly 700-800 T-states including overhead. This is about 1% of a frame budget – negligible.

Stereo Configuration

TurboSound enables true stereo. The two chips can be hard-panned or mixed:

Configuration	Chip 0 (Left)	Chip 1 (Right)	Character
Wide stereo	Lead, bass (A0+B0), drums (C0)	Counter-melody (A1+B1), pads (C1)	Spacious, concert-like
Centered bass	Bass on C0 + C1 (same data)	Lead on A0, harmony on A1	Solid low end

Configuration	Chip 0 (Left)	Chip 1 (Right)	Character
Split drums	Kick (C0)	Snare + hi-hat (C1)	Punchy, wide percussion

Each chip has its own independent noise generator and envelope generator. This means you can run buzz-bass on chip 0 and a completely independent envelope percussion on chip 1 without interference – impossible on a single AY.

ZX Spectrum Next: Triple AY (3 x AY)

The ZX Spectrum Next includes three AY-compatible sound chips, providing 9 tone channels, 3 noise generators, and 3 envelope generators.

Chip Selection on Next

The Next extends the TurboSound protocol. The third chip is selected via the same mechanism:

Chip Select Value	Target
`$FF`	Chip 0
`$FE`	Chip 1
`$FD`	Chip 2

The Next also provides access through Next register $06 (Peripheral 2), which configures the sound chip mode:

Next Reg $06, bits 1-0	Mode
`%00`	Single AY (Spectrum 128K compatible)
`%01`	TurboSound (2 x AY)
`%10`	Triple AY (3 x AY)

Per-Channel Stereo Panning

The Next adds a feature the original AY never had: per-channel stereo panning. Each of the 9 channels can be individually panned left, right, or centre.

Panning is controlled through AY register 14 (R14), repurposed on the Next as a stereo control register when accessed in the AY chip context:

Bit	Channel	0 =	1 =
0	A, right	off	on
1	A, left	off	on
2	B, right	off	on
3	B, left	off	on
4	C, right	off	on

Bit	Channel	0 =	1 =
5	C, left	off	on

Set both bits (left + right) for centre panning. Set only one bit for hard left or hard right.

Common Patterns in Z80 Code

Silence the AY

```
; Silence all channels immediately
ay_silence:
    xor  a                ; volume = 0
    ld   e, a
    ld   a, 8             ; R8 = Volume A
    call ay_write
    ld   a, 9             ; R9 = Volume B
    call ay_write
    ld   a, 10            ; R10 = Volume C
    call ay_write
    ; Disable all tone and noise
    ld   a, 7             ; R7 = Mixer
    ld   e, $3F           ; all off
    call ay_write
    ret
```

Play a Single Note on Channel A

```
; Play note with period DE on channel A at volume 12
; Assumes mixer is already configured for tone A
play_note_a:
    ld   a, 0             ; R0 = Tone A low
    ld   e, d             ; wait -- let's do this properly
    ; Actually: DE has period, E = low byte, D = high byte
    push de
    ld   a, 0             ; R0
    call ay_write          ; E already has low byte
    pop  de
    ld   e, d
    ld   a, 1             ; R1
    call ay_write          ; E = high byte
    ld   a, 8             ; R8 = Volume A
    ld   e, 12            ; volume 12
    call ay_write
    ret
```

Trigger a Buzz-Bass Note

```
; Buzz-bass: envelope-based bass note
```

```
; DE = envelope period for desired pitch
buzz_bass:
    ld   a, 11             ; R11 = Envelope period low
    call ay_write           ; E = low byte of period
    ld   e, d
    ld   a, 12             ; R12 = Envelope period high
    call ay_write
    ld   a, 13             ; R13 = Envelope shape
    ld   e, $08            ; repeating sawtooth down
    call ay_write           ; this also restarts the envelope
    ld   a, 10             ; R10 = Volume C
    ld   e, $10            ; envelope mode
    call ay_write
    ret
```

Trigger Envelope (Restart)

```
; Restart the envelope without changing its shape
; Useful for re-triggering buzz-bass on a new note
; The shape value must be written even if unchanged
env_restart:
    ld   a, 13             ; R13 = Envelope shape
    ld   e, (hl)           ; shape from current instrument data
    call ay_write           ; writing R13 = instant restart
    ret
```

Digital Drum Hit (4-bit DAC)

```
; Play a short digital sample through AY volume register
; HL = pointer to 4-bit sample data (values 0-15)
; B  = number of samples to play
; Uses channel A (R8) as DAC
;
; WARNING: This consumes all CPU time during playback.
; Disable interrupts before calling.
play_sample:
    di
    ld   a, 8              ; select R8 (Volume A)
    ld   bc, $FFFD
    out  (c), a
    ld   c, $FD            ; prepare for $BFFD writes
.loop:
    ld   a, (hl)           ; 7T   load sample
    inc  hl                ; 6T   advance pointer
    ld   b, $BF            ; 7T   BC = $BFFD
    out  (c), a            ; 12T  write volume = sample
    ; Timing padding: adjust NOPs to hit target sample rate
    nop                    ; 4T
    nop                    ; 4T
    ld   b, a              ; 4T   (dummy, preserves timing)
    ld   b, 0              ; 7T   reset B for next iteration
    ; Total per sample: ~51T = ~14.6 us = ~68.6 kHz
```

```
    ; For 8 kHz, add ~386T of padding (96 NOPs or a delay loop)
    dec  b
    jr   nz, .loop
    ei
    ret
```

See Chapter 12 for the hybrid drum technique (digital attack + AY envelope decay) that makes this practical in a demo engine.

Tracker Format Notes

ProTracker 3 (.pt3)

ProTracker 3 (PT3) is the most widely used tracker format on the ZX Spectrum 128K. The cross-platform editor **Vortex Tracker II** natively produces .pt3 files.

Property	Value
Channels	3 (single AY) or 6 (TurboSound)
Pattern length	1-64 rows
Song speed	1-255 (frames per row)
Instruments	Ornament tables + amplitude envelopes
Effects	Arpeggio, portamento, vibrato, envelope control
Player size	~1.2-1.8 KB (Z80 assembly)
Data format	Packed patterns + instrument/ornament tables

How PT3 Data Maps to AY Registers

On each frame (1/50th second), the PT3 player:

1. Decrements the speed counter. If it reaches zero, advance to the next row.
2. For each channel (A, B, C):
   - Read the note, instrument, and effect from the current row.
   - Apply the instrument's ornament (per-frame pitch offset table).
   - Apply the instrument's amplitude envelope (per-frame volume table).
   - Calculate the final tone period and volume.
3. Update the mixer register (R7) based on which channels use tone, noise, or both.
4. Write all calculated values to R0-R13.

```
PT3 row data:
  Note C-4, Instrument 3, Effect: Arpeggio +4+7
       |          |              |
       v          v              v
  Period=424  Ornament table  +4 semitones, +7 semitones
       |      applied per-frame  cycling every 3 frames
       v          |              |
  R0=$A8, R1=$01  |              |
       <----------+              |
  Final period adjusted by ornament offset
       <----------------------------+
```

```
  Period cycles: 424 -> 337 -> 283 -> 424 -> ...
  (C4  -> E4  -> G4  -> C4  -> ...)
```

Integration in Your Project

```
; Standard PT3 integration
; 1. Include the player source
    INCLUDE "pt3player.asm"

; 2. Include the song data
song_data:
    INCBIN "mysong.pt3"

; 3. Initialise at startup
    ld   hl, song_data
    call music_init         ; PT3 player init routine

; 4. Call from IM2 handler every frame
isr_handler:
    ; ... push registers ...
    call music_play         ; PT3 player frame routine
    ; ... pop registers ...
    ei
    reti
```

Other Tracker Formats

Format	Tracker	Channels	Player Size	Key Feature
.pt3	Pro Tracker 3 / Vortex Tracker II	3-6	~1.2-1.8 KB	Industry standard. Ornaments + samples.
.pt2	Pro Tracker 2	3	~0.8-1.0 KB	Older, simpler. Still used for size reasons.
.stc	Sound Tracker / Sound Tracker Pro	3	~0.5-0.7 KB	Oldest Spectrum format. No ornaments.
.asc	ASC Sound Master	3	~0.8-1.0 KB	Compact player. Russian scene favourite.
.sqt	SQ-Tracker	3	~0.6-0.8 KB	Excellent data compression.
.ay	AY emulation container	varies	N/A	Captures raw register dumps. For emulators, not playback engines.

Vortex Tracker II

Vortex Tracker II is the modern standard for composing AY music. It runs on Windows (and via Wine on Linux/macOS), and directly exports .pt3 files compatible with all standard Z80 players.

Key features for demoscene use: - **TurboSound mode:** Edit 6 channels (2 x AY) in a single module. - **Ornament editor:** Visual per-frame pitch offset tables. - **Sample editor:** Per-frame amplitude + tone/noise control. - **Loop points:** Set loop start for each pattern and for the entire song. - **Export:** .pt3 (native), .txt (plain text dump for analysis), .wav (audio render).

The typical demoscene workflow: 1. Compose in Vortex Tracker II. 2. Export as .pt3. 3. Include the standard `pt3player.asm` (widely available, multiple versions optimised for size or speed). 4. `INCBIN` the .pt3 data. 5. Call `music_init` and `music_play` as shown above.

AY-3-8910 vs YM2149: Differences That Matter

The Yamaha YM2149 (used in the Atari ST and some Spectrum clones) is pin-compatible with the AY-3-8910 but not bit-identical:

Feature	AY-3-8910	YM2149
Volume curve	Logarithmic (1.5 dB/step)	Linear
Noise LFSR	17-bit, specific taps	17-bit, different taps
Envelope precision	16 volume steps (4-bit)	32 volume steps (5-bit internally)
Pin 26 (SEL)	Clock divider	Same, but often hard-wired
Output DAC	4-bit resistor ladder	5-bit resistor ladder

Practical impact: The same tune sounds warmer/bassier on a real AY-3-8910 and brighter/thinner on a YM2149 due to the different volume curves. Emulators typically let you select which chip to emulate. When testing your music, try both – your audience may have either chip in their machine.

Quick Reference Card

Register Summary (Tear-Out)

```
R0  = Tone A low        (8 bits)    R/W
R1  = Tone A high       (4 bits)    R/W
R2  = Tone B low        (8 bits)    R/W
R3  = Tone B high       (4 bits)    R/W
R4  = Tone C low        (8 bits)    R/W
R5  = Tone C high       (4 bits)    R/W
R6  = Noise period      (5 bits)    R/W    0=highest, 31=lowest
```

```
R7  = Mixer             (8 bits)    R/W    0=ON (active low!)
R8  = Volume A          (5 bits)    R/W    bit4=envelope mode
R9  = Volume B          (5 bits)    R/W    bit4=envelope mode
R10 = Volume C          (5 bits)    R/W    bit4=envelope mode
R11 = Envelope low      (8 bits)    R/W
R12 = Envelope high     (8 bits)    R/W
R13 = Envelope shape    (4 bits)    W      write = restart!

Ports:  $FFFD = select register    $BFFD = write data
Write:  OUT ($FFFD),reg  then  OUT ($BFFD),value
Read:   OUT ($FFFD),reg  then  IN A,($FFFD)

Tone:   freq = 1773400 / (16 * period)      period = 12-bit (1-4095)
Env:    freq = 1773400 / (256 * period)      period = 16-bit (1-65535)
Noise:  freq = 1773400 / (16 * period)       period = 5-bit (0-31)

Mixer R7 bits:  [IOB IOA | NC NB NA | TC TB TA]   0=enable, 1=disable
Envelope R13:   [CONT ATT ALT HOLD]
  $00 = \___    $08 = \\\\    $0A = \/\/    $0B = \^^^
  $04 = /___    $0C = ////    $0E = /\/\    $0D = /^^^

TurboSound:  $FF->$FFFD = chip 0    $FE->$FFFD = chip 1
Next 3xAY:   $FF = chip 0   $FE = chip 1   $FD = chip 2
```

Sources: General Instrument AY-3-8910 / AY-3-8912 datasheet (1979); Yamaha YM2149 Application Manual; Dark “GS Sound System” (Spectrum Expert #01, 1997); Introspec “Eager” making-of (Hype 2015); Vortex Tracker II documentation (S.V. Bulba); NedoPC TurboSound FM documentation; ZX Spectrum Next User Manual, Issue 2

Appendix H: Storage APIs — TR-DOS and esxDOS

> *"A technically impressive demo that ships as a .tzx when the rules require .trd will be disqualified."* – Chapter 20

Two storage APIs dominate the ZX Spectrum world: **TR-DOS** (the disk operating system of the Soviet Beta Disk 128 interface, standard on Pentagon and Scorpion clones) and **esxDOS** (the modern SD card operating system running on DivMMC and DivIDE hardware). Most Russian and Ukrainian demoscene releases ship as `.trd` disk images. Most modern Western releases use `.tap` tape images or esxDOS-compatible file layouts. If you are releasing a demo or game today, the practical choice is to provide a `.trd` image for compatibility with the enormous Russian/Ukrainian install base, and a `.tap` file for everyone else. If your loader supports esxDOS detection (as described in Chapter 21), users with DivMMC hardware get fast SD card loading for free.

This appendix is the API reference you keep open while writing your loader. Chapter 21 covers integration in a full game project. Chapter 15 covers the hardware details of memory banking and port mapping that underpin both APIs.

---

1. TR-DOS (Beta Disk 128)

Hardware

The Beta Disk 128 interface is the standard floppy disk controller for Pentagon, Scorpion, and most Soviet ZX Spectrum clones. It is based on the Western Digital WD1793 floppy disk controller chip, which communicates with the Z80 through five I/O ports.

The TR-DOS ROM (8 KB) occupies `$0000–$3FFF` when the Beta Disk interface is active. It is paged in automatically when the Z80 executes code at address `$3D13` (the magic entry point), and paged out when execution returns to the main ROM area.

Disk Format

Property	Value
Tracks	80
Sides	2

Property	Value
Sectors per track	16
Bytes per sector	256
Total capacity	640 KB (655,360 bytes)
Image format	`.trd` (raw disk image, 640 KB)
System track	Track 0, side 0

Track 0 contains the disk directory (sectors 1–8) and the disk information sector (sector 9). The directory holds up to 128 file entries. Each entry is 16 bytes:

```
Bytes 0-7:   Filename (8 characters, space-padded)
Byte  8:     File type: 'C' = code, 'B' = BASIC, 'D' = data, '#' = sequential
Bytes 9-10:  Start address (or BASIC line number)
Bytes 11-12: Length in bytes
Byte  13:    Length in sectors
Byte  14:    Starting sector
Byte  15:    Starting track
```

WD1793 Port Map

Port	Read	Write
`$1F`	Status register	Command register
`$3F`	Track register	Track register
`$5F`	Sector register	Sector register
`$7F`	Data register	Data register
`$FF`	TR-DOS system register	TR-DOS system register

Port `$FF` is the Beta Disk system port. It controls drive selection, side selection, head load, and density. The upper bits also reflect the DRQ (Data Request) and INTRQ (Interrupt Request) signals from the WD1793.

WD1793 Commands

Command	Code	Description
Restore	`$08`	Move head to track 0. Verify track.
Seek	`$18`	Move head to track in data register.
Step In	`$48`	Step head one track toward centre.
Step Out	`$68`	Step head one track toward edge.
Read Sector	`$88`	Read one 256-byte sector.
Write Sector	`$A8`	Write one 256-byte sector.
Read Address	`$C0`	Read the next sector ID field.
Force Interrupt	`$D0`	Abort current command.

The low nibble of each command byte carries modifier flags (step rate, verify, side select, delay). The values above use common defaults. Consult the WD1793 datasheet for the full bit layout.

ROM API: Loading a File

The standard approach to file I/O under TR-DOS is to call the ROM routines at `$3D13`. The TR-DOS ROM provides high-level file operations through a command system: you place parameters in registers and in the system area at `$5D00–$5FFF`, then call into the ROM.

```
; TR-DOS: Load a file by name
; Loads a code file ('C' type) to its stored start address
;
; The filename must be placed at $5D02 (8 bytes, space-padded).
; The file type goes to $5D0A.
;
; Call $3D13 with C = $08 (load file command)

load_trdos_file:
    ; Set up filename at TR-DOS system area
    ld   hl, my_filename
    ld   de, $5D02
    ld   bc, 8
    ldir                    ; copy 8-char filename

    ld   a, 'C'             ; file type: code
    ld   ($5D0A), a

    ld   c, $08             ; TR-DOS command: load file
    call $3D13              ; enter TR-DOS ROM
    ret

my_filename:
    db   "SCREEN  "         ; 8 characters, space-padded
```

To load to a specific address (overriding the stored start address):

```
; TR-DOS: Load file to explicit address
; HL = destination address
; DE = length to load
; Filename already at $5D02, type at $5D0A

load_trdos_to_addr:
    ld   hl, $4000          ; load to screen memory
    ld   de, 6912           ; 6912 bytes (one screen)
    ld   ($5D03), hl        ; override start address
    ld   ($5D05), de        ; override length
    ld   c, $08             ; load file
    call $3D13
    ret
```

Direct Sector Access

For demos that stream data from disk — fullscreen animations, music data that exceeds available RAM, or multipart demos that load effects on the fly — direct sector access bypasses the file system entirely. You control the head position, read sectors one at a time, and process the data as it arrives.

```
; Read a single sector directly via WD1793 ports
; B = track number (0-159, with side encoded in bit 0 of $FF)
; C = sector number (1-16)
; HL = destination buffer (256 bytes)

read_sector:
    ld   a, b
    out  ($3F), a           ; set track register
    ld   a, c
    out  ($5F), a           ; set sector register

    ld   a, $88             ; Read Sector command
    out  ($1F), a           ; issue command

    ; Wait for DRQ and read 256 bytes
    ld   b, 0               ; 256 bytes to read
.wait_drq:
    in   a, ($FF)           ; read system register
    bit  6, a               ; test DRQ bit
    jr   z, .wait_drq       ; wait until data ready
    in   a, ($7F)           ; read data byte
    ld   (hl), a
    inc  hl
    djnz .wait_drq

    ; Wait for command completion
.wait_done:
    in   a, ($1F)           ; read status register
    bit  0, a               ; test BUSY bit
    jr   nz, .wait_done
    ret
```

Warning: Direct sector access is timing-sensitive. Interrupts must be disabled during the data transfer loop, or bytes will be lost. The WD1793 asserts DRQ for a limited time window; if the Z80 does not read the data register before the next byte arrives, data is overwritten. At 250 kbit/s (double density), you have approximately 32 microseconds per byte — about 112 T-states on a Pentagon. The tight loop above runs in roughly 50–60 T-states per byte, leaving adequate margin.

Disk Detection

To detect whether a Beta Disk interface is present:

```
; Detect Beta Disk 128
; Returns: carry clear if present, carry set if absent
detect_beta_disk:
    ; The TR-DOS ROM signature is at $0069 when paged in.
    ; We can check port $FF for a sane response:
    ; If no Beta Disk is present, port $FF reads as floating bus.
    in   a, ($1F)           ; read WD1793 status
    cp   $FF                ; floating bus returns $FF
    scf
    ret  z                  ; probably no controller
```

```
    or   a                  ; clear carry = present
    ret
```

A more robust method is to attempt to call `$3D13` and check if TR-DOS ROM signature bytes are present. Production code typically checks for a known byte sequence at the TR-DOS ROM entry points.

---

2. esxDOS (DivMMC / DivIDE)

Hardware

DivMMC (and its older sibling DivIDE) is a mass storage interface that connects an SD card to the ZX Spectrum. The esxDOS firmware provides a POSIX-like file API accessible from Z80 code through `RST $08`. esxDOS supports FAT16 and FAT32 file systems, long filenames, subdirectories, and multiple open file handles.

DivMMC uses auto-mapping: when the Z80 fetches an instruction from certain "trap" addresses (notably `$0000`, `$0008`, `$0038`, `$0066`, `$04C6`, `$0562`), the DivMMC hardware automatically pages in its own ROM at `$0000–$1FFF`. The `RST $08` trap is the primary API entry point.

API Pattern

Every esxDOS call follows the same pattern:

```
    rst  $08              ; trigger DivMMC auto-map
    db   function_id      ; function number (byte after RST)
    ; Returns:
    ;   Carry clear = success
    ;   Carry set   = error, A = error code
```

The function number is the byte immediately following the `RST $08` instruction in memory. The Z80 executes `RST $08`, which jumps to address `$0008`. DivMMC auto-maps its ROM at that address, reads the next byte (the function number), dispatches the call, then un-maps its ROM and returns to the instruction after the `DB`.

Function Reference

Function	ID	Description	Input	Output
`M_GETSETDRV`	`$89`	Get/set default drive	A = `'*'` for default	A = drive letter
`F_OPEN`	`$9A`	Open file	IX = filename (zero-terminated), B = mode, A = drive	A = file handle
`F_CLOSE`	`$9B`	Close file	A = file handle	–

Function	ID	Description	Input	Output
F_READ	$9D	Read bytes	A = handle, IX = buffer, BC = count	BC = bytes read
F_WRITE	$9E	Write bytes	A = handle, IX = buffer, BC = count	BC = bytes written
F_SEEK	$9F	Seek in file	A = handle, L = whence, BCDE = offset	BCDE = new position
F_FSTAT	$A1	File status (by handle)	A = handle, IX = buffer	11-byte stat block
F_OPENDIR	$A3	Open directory	IX = path (zero-terminated)	A = dir handle
F_READDIR	$A4	Read directory entry	A = dir handle, IX = buffer	entry at (IX)
F_CLOSEDIR	$A5	Close directory	A = dir handle	–
F_GETCWD	$A8	Get current directory	IX = buffer	path at (IX)
F_CHDIR	$A9	Change directory	IX = path	–
F_STAT	$AC	File status (by name)	IX = filename	11-byte stat block

File Open Modes

Mode	Value	Description
Read only	$01	Open existing file for reading
Create/truncate	$06	Create new or truncate existing for writing
Create new only	$04	Create new file; fail if exists
Append	$0E	Open for writing at end of file

Seek Whence Values

Whence	Value	Description
SEEK_SET	$00	Offset from beginning of file
SEEK_CUR	$01	Offset from current position
SEEK_END	$02	Offset from end of file

Code Example: Load a File

```
; esxDOS: Load a binary file into memory
;
```

```
; Uses register conventions from esxDOS API documentation.
; Note: F_READ uses IX for the destination buffer, not HL.

    ld   a, '*'             ; use default drive
    ld   ix, filename       ; pointer to zero-terminated filename
    ld   b, $01             ; FA_READ: open for reading
    rst  $08
    db   $9A                ; F_OPEN
    jr   c, .error          ; carry set = error

    ld   (.file_handle), a  ; save file handle

    ld   ix, $4000          ; destination buffer (screen memory)
    ld   bc, 6912           ; bytes to read (one full screen)
    ld   a, (.file_handle)
    rst  $08
    db   $9D                ; F_READ
    jr   c, .error

    ld   a, (.file_handle)
    rst  $08
    db   $9B                ; F_CLOSE
    ret

.error:
    ; A contains the esxDOS error code
    ; Common errors:
    ;   5 = file not found
    ;   7 = file already exists (on create)
    ;   9 = invalid file handle
    ret

filename:
    db   "screen.scr", 0

.file_handle:
    db   0
```

Code Example: Streaming Data from File

For demos that load data incrementally — decompressing level chunks between frames, streaming a pre-rendered animation, or loading music patterns on demand — the pattern is: open the file once, read a chunk per frame, close when done.

```
; Streaming: read N bytes per frame from an open file
; Call stream_init once, then stream_chunk from your main loop.

CHUNK_SIZE  equ  256        ; bytes per frame (tune to budget)

stream_handle:  db 0
stream_done:    db 0

; Initialise: open the file
```

```
stream_init:
    ld   a, '*'
    ld   ix, stream_file
    ld   b, $01             ; FA_READ
    rst  $08
    db   $9A                ; F_OPEN
    ret  c                  ; error
    ld   (stream_handle), a
    xor  a
    ld   (stream_done), a   ; not done yet
    ret

; Per-frame: read one chunk into buffer
; Returns: BC = bytes actually read (may be < CHUNK_SIZE at EOF)
stream_chunk:
    ld   a, (stream_done)
    or   a
    ret  nz                 ; already finished

    ld   a, (stream_handle)
    ld   ix, stream_buffer
    ld   bc, CHUNK_SIZE
    rst  $08
    db   $9D                ; F_READ
    jr   c, .eof

    ; BC = bytes actually read
    ld   a, b
    or   c
    jr   z, .eof            ; zero bytes read = end of file
    ret

.eof:
    ld   a, (stream_handle)
    rst  $08
    db   $9B                ; F_CLOSE
    ld   a, 1
    ld   (stream_done), a
    ret

stream_file:
    db   "anim.bin", 0

stream_buffer:
    ds   CHUNK_SIZE
```

Detecting esxDOS

```
; Detect esxDOS presence
; Returns: carry clear = esxDOS available, carry set = not available
;
; Strategy: attempt M_GETSETDRV. If esxDOS is present, it returns
```

```
; the current drive letter. If not present, RST $08 goes to the
; Spectrum ROM's error handler at $0008 (a benign instruction on
; the 128K ROM) and does not crash.

detect_esxdos:
    ld   a, '*'             ; request default drive
    rst  $08
    db   $89                ; M_GETSETDRV
    ret                     ; carry flag set by esxDOS on error
```

A more conservative approach checks for the DivMMC's trap signature before calling any API functions. In practice, the method above works on all 128K models because the 128K ROM's `$0008` handler does not crash — it executes a benign sequence and returns. On a 48K machine without esxDOS, `RST $08` goes to the error restart, which may need special handling. Chapter 21 discusses this in the context of a production game loader.

---

3. +3DOS (Amstrad +3)

The Amstrad Spectrum +3, with its built-in 3-inch floppy drive, has its own DOS: +3DOS. The API uses a different mechanism — calls to entry points in the +3DOS ROM at page `$01`, accessed through `RST $08` with a different set of function codes.

+3DOS is rarely used in the demoscene for two reasons. First, the +3 was primarily sold in Western Europe and was never the dominant Spectrum model in any scene community. Second, the +3's non-standard memory layout and ROM paging scheme make it incompatible with most demoscene code written for the 128K/Pentagon architecture. If you need +3 compatibility, the +3DOS API is documented in the Spectrum +3 technical manual (Amstrad, 1987). For most demo and game projects, providing a `.tap` file is sufficient — the +3 loads `.tap` files natively through its tape compatibility mode.

---

4. Practical Patterns

Loading Screen from Disk (TR-DOS)

The loading screen is the user's first impression. On TR-DOS, the screen file (`SCREEN C`, 6912 bytes) loads directly to `$4000` and appears immediately:

```
; TR-DOS: Load a .scr file directly to screen memory
; The screen appears as it loads, line by line.
load_screen_trdos:
    ld   hl, scr_filename
    ld   de, $5D02
    ld   bc, 8
    ldir
    ld   a, 'C'
    ld   ($5D0A), a
    ld   hl, $4000          ; destination: screen memory
```

```
    ld   ($5D03), hl
    ld   de, 6912           ; length: full screen
    ld   ($5D05), de
    ld   c, $08             ; load file
    call $3D13
    ret

scr_filename:
    db   "SCREEN  "         ; 8 chars, padded
```

Loading Screen from SD (esxDOS)

Same visual result, different API:

```
; esxDOS: Load a .scr file to screen memory
load_screen_esxdos:
    ld   a, '*'
    ld   ix, scr_filename_esx
    ld   b, $01             ; FA_READ
    rst  $08
    db   $9A                ; F_OPEN
    ret  c

    push af                 ; save handle
    ld   ix, $4000          ; destination: screen memory
    ld   bc, 6912
    pop  af
    push af
    rst  $08
    db   $9D                ; F_READ
    pop  af
    rst  $08
    db   $9B                ; F_CLOSE
    ret

scr_filename_esx:
    db   "screen.scr", 0
```

Dual-Mode Loader

A production loader should detect the available storage and use it:

```
; Unified loader: try esxDOS first, fall back to TR-DOS, then tape
load_data:
    call detect_esxdos
    jr   nc, .use_esxdos    ; carry clear = esxDOS present

    call detect_beta_disk
    jr   nc, .use_trdos     ; carry clear = Beta Disk present

    ; Fall back to tape loading
    jp   load_from_tape
```

```
.use_esxdos:
    jp   load_from_esxdos

.use_trdos:
    jp   load_from_trdos
```

Streaming Compressed Data

The most powerful pattern combines storage API with compression (Appendix C). Open a file containing compressed data, read chunks into a buffer each frame, decompress into the destination, and advance:

```
Frame 1:  F_READ 256 bytes -> buffer   |  decompress buffer -> screen
Frame 2:  F_READ 256 bytes -> buffer   |  decompress buffer -> screen
Frame 3:  F_READ 256 bytes -> buffer   |  decompress buffer -> screen
...
Frame N:  F_READ < 256 bytes (EOF)     |  decompress, close file
```

At 256 bytes per frame and 50 fps, you stream 12.5 KB/sec from the SD card — enough for a compressed fullscreen animation. On TR-DOS, direct sector reads at one sector per frame give 12.8 KB/sec (256 bytes * 50 fps). The bottleneck is decompression speed, not I/O.

---

5. File Format Reference

Format	Extension	Usage	Notes
TR-DOS disk image	`.trd`	Standard for Pentagon/Scorpion releases	Raw 640 KB image. Every emulator supports it.
TR-DOS file container	`.scl`	Simpler than .trd	Contains files without full disk structure. Good for distribution.
Tape image	`.tap`	Universal tape format	Works on every Spectrum model and emulator. No file system.
Extended tape image	`.tzx`	Tape with copy protection / turbo loaders	Preserves exact tape timing. Rarely needed for new releases.
Snapshot (48K/128K)	`.sna`	Quick load, no file system	Captures full machine state. No loading code needed.

Format	Extension	Usage	Notes
Snapshot (compressed)	`.z80`	Like .sna but compressed	Multiple versions; .z80 v3 supports 128K.
Next distribution	`.nex`	ZX Spectrum Next executable	Self-contained binary with header specifying bank layout.

Choosing a release format: For a demoscene release, provide at least two formats:

1. **`.trd`** for TR-DOS users (the Russian/Ukrainian community, Pentagon/Scorpion owners, and emulator users who prefer disk images). This is the default for parties like Chaos Constructions and DiHalt.
2. **`.tap`** for everyone else (real 128K hardware with tape input, DivMMC users via `.tap` loader, and all emulators). sjasmplus can generate `.tap` output directly with its `SAVETAP` directive.

If your demo is small enough (under 48 KB), a `.sna` snapshot also works well — it loads instantly with no loader code.

---

6. See Also

- **Chapter 15** — Hardware Anatomy: memory banking, port `$7FFD`, the full port map that TR-DOS and esxDOS sit alongside.
- **Chapter 20** — Demo Workflow: release formats, party submission rules, `.trd` vs `.tap` requirements.
- **Chapter 21** — Full Game: production-quality tape and esxDOS loading code, dual-mode detection, bank-by-bank loading.
- **Appendix C** — Compression: which compressors to pair with streaming I/O.
- **Appendix E** — eZ80 / Agon Light 2: the MOS file API on Agon, which provides similar file operations (`mos_fopen`, `mos_fread`, `mos_fclose`) through a different mechanism (RST $08 with MOS function codes in ADL mode).

Sources: WD1793 datasheet (Western Digital, 1983); TR-DOS v5.03 disassembly (various, public domain); esxDOS API documentation (Wikipedia, zxe.io); DivMMC hardware specification (Mario Prato / ByteDelight); Spectrum +3 Technical Manual (Amstrad, 1987); Introspec, “Loading and saving on the Spectrum” (Hype, 2016)

Appendix I: Bytebeat and AY-Beat – Generative Sound on Z80

> *"At 256 bytes, bytebeat is your only realistic option – there is no room for a pattern player."* – Chapter 13

This appendix covers formula-driven sound generation on the ZX Spectrum – from the original PCM bytebeat concept to the AY-adapted technique that produces structured, evolving music from a handful of Z80 instructions. Chapter 13 introduces AY-beat as a size-coding tool. This appendix is the full reference: the theory, the formulas, the register mappings, and a complete working engine you can drop into a 256-byte intro.

You will need the AY register reference in Appendix G open alongside this appendix. Every register number mentioned here (R0, R7, R8, R11, R13, etc.) is documented there with full bit layouts and port addresses.

---

1. Classic Bytebeat: The PCM Tradition

In 2011, Ville-Matias Heikkila (Viznut) published a discovery that had been circulating in underground programming circles: a single C expression, evaluated once per sample with an incrementing counter `t`, can produce complex rhythmic music when the output is interpreted as 8-bit unsigned PCM at 8 kHz.

The core idea:

```
for (t = 0; ; t++)
    putchar( f(t) );    // pipe to /dev/dsp at 8000 Hz
```

The function `f(t)` is typically a one-line expression built from bitwise operations, multiplication, and bit shifts. No oscillators, no envelopes, no note tables – just integer arithmetic on a counter.

Famous Formulas

`t*((t>>12|t>>8)&63&t>>4)` – Viznut's original. Cascading rhythmic tones that cycle through pitch relationships, producing an effect somewhere between a music box and a broken telephone. The `t>>12` and `t>>8` create two frequency-divided versions

of the counter; `&63` limits the range; `&t>>4` gates the output rhythmically. The multiplication by `t` creates the fundamental pitch sweep.

`t*(t>>5|t>>8)>>(t>>16)` – Evolving rhythmic patterns. The right-shift by `t>>16` means the entire character of the sound changes every ~8 seconds (65536 samples at 8 kHz). Each 8-second section has a different dynamic range and feel.

`(t*5&t>>7)|(t*3&t>>10)` – Two interleaved melodic lines. The `t*5` and `t*3` create two pitch streams at different intervals; the AND with shifted counters gates them independently; the OR merges them. The result sounds like two interlocking melodies playing simultaneously.

Why It Works

Bitwise operations on an incrementing counter create periodic structures at multiple time scales simultaneously. Consider the bit pattern of `t` as it counts:

- Bit 0 toggles every sample (4000 Hz – inaudible as pitch, but shapes the waveform)
- Bit 7 toggles every 128 samples (~62.5 Hz – bass territory)
- Bit 12 toggles every 4096 samples (~1.95 Hz – rhythmic pulse)
- Bit 15 toggles every 32768 samples (~0.24 Hz – structural change)

A right-shift `t>>n` selects which time scale dominates. AND operations create coincidence patterns – moments when two time scales align. OR operations merge independent patterns. Multiplication by small constants creates harmonic relationships (frequency ratios). The 8-bit truncation of the output acts as a natural waveform shaper, folding values back into range and creating additional harmonics.

The result is self-similar: the sound has rhythmic structure at every scale, from individual oscillation cycles up to multi-second phrase structures. This self-similarity is what makes bytebeat sound like music rather than noise – even though no musical knowledge went into the formula.

On the Spectrum: The Beeper Dead End

The ZX Spectrum's beeper output is a 1-bit speaker controlled by bit 4 of port $FE. You can, in principle, run a bytebeat formula and output the result:

```
; Beeper bytebeat -- uses all CPU, no visuals possible
; DE = t (16-bit counter)
    ld   de, 0
.loop:
    ; Compute f(t) -- simplified: A = t AND (t >> 8)
    ld   a, e          ; 4T   A = low byte of t
    and  d             ; 4T   A = t_lo AND t_hi
    ; Output bit 4 to speaker
    and  $10           ; 7T   isolate bit 4
    out  ($FE), a      ; 11T  toggle speaker
    inc  de            ; 6T   t++
    jr   .loop         ; 12T
                       ; --- 44T per sample = ~79.5 kHz
```

This runs, and it produces sound. But it consumes 100% of the CPU – the Z80 is doing nothing but computing samples and toggling the speaker. No screen updates,

no visual effects, no input handling. The sample rate is also wrong (too fast), and controlling it precisely requires careful cycle counting with padding NOPs.

For a demo, this is a dead end. The beeper is a 1-bit output that demands constant CPU attention. The real adaptation of bytebeat to the Spectrum requires a different approach entirely.

---

2. AY-Beat: Bytebeat Reimagined for a Tone Generator

The AY-3-8910 is not a DAC. It does not accept amplitude samples. It is a programmable tone generator: you give it a frequency (as a period value), a volume (0-15), and optional noise and envelope parameters, and its internal oscillators produce the sound autonomously. The CPU is free to do other work.

The key insight of AY-beat: **replace the sample counter with a frame counter, and replace PCM output with AY register values.**

Classic bytebeat computes one amplitude sample at ~8000 Hz. AY-beat computes tone periods, volumes, and noise parameters at 50 Hz – once per video frame, triggered by the HALT instruction. The AY's oscillators handle the actual sound generation between frames.

The frame counter `t` replaces the sample counter. Formulas operate on `t` but produce register values, not waveform samples. Where PCM bytebeat has one degree of freedom (amplitude), AY-beat has many: three independent tone periods (12-bit each), three volumes (4-bit each), one noise period (5-bit), and a 16-bit envelope period with shape selection.

Basic AY-Beat Architecture

```
; AY-beat frame update -- called once per HALT
; Assumes frame_counter is a byte in memory
ay_beat_update:
    ld   a, (frame_counter)
    ld   e, a             ; E = t (keep a copy)

    ; === Channel A: tone period from formula ===
    ; tone_lo = (t * 3) AND $3F -- pentatonic-ish cycling
    add  a, a             ; 4T   A = t * 2
    add  a, e             ; 4T   A = t * 3
    and  $3F              ; 7T   mask to 6 bits (periods 0-63)
    ld   d, a             ; 4T   save tone period
    ; Write R0 (tone A low) = D
    xor  a                ; 4T   A = 0 (register number)
    call ay_write_d       ; writes D to AY register A

    ; Write R1 (tone A high) = 0
    ld   a, 1             ; 7T
    ld   d, 0             ; 7T
    call ay_write_d
```

```
    ; === Channel A: volume from formula ===
    ; volume = bits 6-3 of t, giving 0-15 cycling
    ld   a, e             ; 4T   reload t
    rrca                  ; 4T
    rrca                  ; 4T
    rrca                  ; 4T
    and  $0F              ; 7T   volume 0-15
    ld   d, a             ; 4T
    ld   a, 8             ; 7T   R8 = Volume A
    call ay_write_d

    ; Advance frame counter
    ld   hl, frame_counter
    inc  (hl)             ; 11T
    ret
```

This is the simplest possible AY-beat: one channel, one tone formula, one volume formula, ~30 bytes. It produces a cycling sweep that rises in pitch and fades in and out – not music, but recognisably structured sound.

What Changes from PCM Bytebeat

Aspect	Classic (PCM)	AY-Beat
Update rate	~8000 Hz	50 Hz (frame rate)
Output	8-bit amplitude	Tone period (12-bit), volume (4-bit), noise (5-bit)
Channels	1 (mono speaker)	3 tone + 1 noise + envelope
CPU cost	100% (all cycles)	~200-500 T per frame (~0.3%)
Formula scaling	Fine-grained, fast evolution	Coarse-grained, need wider bit shifts
Sound generation	CPU computes every sample	AY hardware oscillators run autonomously

The 50 Hz frame rate means formulas evolve 160x slower than at 8 kHz. To get equivalent rhythmic density, use larger multipliers and fewer right-shifts. A formula that produces a pleasant rhythm at 8 kHz with `t>>12` (period ~0.5 sec at 8 kHz) needs approximately `t>>4` at 50 Hz for similar timing (~0.3 sec between repetitions). The general rule: divide the PC bytebeat shift amounts by ~7 (log2 of the 160x ratio) and adjust by ear.

---

3. Drone: Envelope + Tone (E+T Mode)

This is where AY-beat gets genuinely interesting. The AY envelope generator automatically cycles the volume of a channel without any CPU intervention. Set a

channel's volume register to envelope mode (bit 4 = 1, i.e. write $10 to R8/R9/R10), and the hardware handles volume modulation at the envelope frequency defined by R11-R12.

The result is a drone: a continuously evolving timbre produced by the interaction of the tone oscillator and the envelope oscillator. The CPU cost for maintaining this drone is almost zero – you only need to update the tone period and envelope period once per frame, and the hardware does the rest.

The Drone Recipe

1. Set a tone period from a formula – this defines the base pitch.
2. Set an envelope period from a different formula – this defines the modulation rate.
3. Set the envelope shape to a repeating waveform (shapes $08, $0A, $0C, or $0E).
4. Set the channel volume to envelope mode ($10).
5. The hardware produces a continuously evolving drone with zero per-sample CPU cost.

```
; Drone setup -- E+T mode
; Tone period evolves per frame, envelope period evolves slower
drone_update:
    ld   a, (frame_counter)
    ld   e, a             ; E = t

    ; --- Tone period: slowly sweeping ---
    and  $7F              ; 128-frame cycle (2.56 seconds)
    ld   d, a
    xor  a                ; R0 = Tone A low
    call ay_write_d
    ld   a, 1             ; R1 = Tone A high
    ld   d, 0
    call ay_write_d

    ; --- Envelope period: evolves on different time scale ---
    ld   a, e             ; reload t
    rrca
    rrca                  ; divide by 4 -- envelope evolves 4x slower
    and  $3F
    add  a, $10           ; offset to avoid very fast envelopes
    ld   d, a
    ld   a, 11            ; R11 = Envelope period low
    call ay_write_d
    ld   a, 12            ; R12 = Envelope period high
    ld   d, 0
    call ay_write_d

    ; --- Envelope shape: repeating triangle ---
    ; CAUTION: writing R13 restarts the envelope cycle.
    ; Only write on the first frame, or when you want a restart.
    ld   a, e
    or   a
```

```
    jr   nz, .skip_shape
    ld   a, 13            ; R13 = Envelope shape
    ld   d, $0A           ; shape $0A = repeating triangle \/\/
    call ay_write_d
.skip_shape:

    ; --- Volume A: envelope mode ---
    ld   a, 8             ; R8 = Volume A
    ld   d, $10           ; bit 4 set = use envelope
    call ay_write_d

    ret
```

The beauty of E+T mode is the interference between the two frequencies. When the envelope period is close to the tone period, you get amplitude modulation effects – the volume beats at the difference frequency, producing a wavering, organ-like timbre. When the envelope frequency is much lower than the tone frequency, it acts as a slow tremolo. When it is much higher, you get buzz-bass territory (see Appendix G and Chapter 11).

Sweeping the envelope period while the tone period also moves produces continuously evolving textures. The two formulas create a two-dimensional parameter space that the sound explores over time. With the right formula pair, the drone never quite repeats – it wanders through timbral variations, creating an ambient soundscape from fewer than 30 bytes of code.

Byte Cost

Setting up E+T mode with a formula takes approximately 15-25 bytes. For a 256-byte intro, this gives you rich, evolving drone sound essentially for free – no per-frame volume computation needed, no pattern data, just two register values derived from simple formulas. The AY hardware is doing all the oscillation work.

---

4. Noise Percussion

The AY noise generator (R6) produces pseudo-random noise at a programmable frequency (0-31). The mixer register (R7) controls which channels receive noise. Toggling noise on and off rhythmically, driven by the frame counter, creates percussion patterns.

Basic Kick Pattern

```
; Noise percussion on channel C
; Frame counter in A (already loaded)
    ld   e, a             ; E = t
    and  $07              ; every 8th frame = 6.25 Hz pulse
    jr   nz, .decay

    ; --- Hit: enable noise on C, max volume ---
    ld   a, 7             ; R7 = Mixer
```

```
    ld   d, %00100100     ; tone C off, noise C on, others unchanged
    call ay_write_d
    ld   a, 6             ; R6 = Noise period
    ld   d, 2             ; low period = harsh, punchy
    call ay_write_d
    ld   a, 10            ; R10 = Volume C
    ld   d, $0F           ; maximum volume
    call ay_write_d
    jr   .done

.decay:
    ; --- Decay: reduce volume each frame ---
    ; Simple approach: volume = 15 - (t AND 7)
    ld   a, e
    and  $07              ; frames since last hit
    ld   d, a
    ld   a, $0F
    sub  d                ; volume = 15 - elapsed
    jr   nc, .vol_ok
    xor  a                ; clamp to 0
.vol_ok:
    ld   d, a
    ld   a, 10            ; R10 = Volume C
    call ay_write_d

.done:
```

Percussion Character by Noise Period

R6 Value	Character	Use
0-3	Harsh click, punchy	Kick drum, rimshot
4-8	Crisp hiss	Snare body
10-15	Broad noise	Open hi-hat
20-31	Low rumble	Distant thunder, ambient

Rhythmic Variety from Bit Masks

Different AND masks on the frame counter produce different rhythmic densities:

Mask	Period	Frequency	Character
`AND $03`	Every 4 frames	12.5 Hz	Rapid fire, hi-hat
`AND $07`	Every 8 frames	6.25 Hz	Standard kick
`AND $0F`	Every 16 frames	3.125 Hz	Half-time, sparse
`AND $1F`	Every 32 frames	1.5625 Hz	Slow pulse, intro

Combine two masks for polyrhythm: test `t AND $07` for the kick, `t AND $03` for the hi-hat. This costs about 10 extra bytes but adds significant rhythmic complexity.

Using the Envelope for Drum Decay

Instead of manually decaying the volume each frame, use the AY envelope generator in single-shot mode. Set R13 to shape $00 (decay to zero, hold), and the hardware handles the volume fade automatically:

```
; Envelope-based drum hit -- zero CPU cost for decay
    ld   a, e
    and  $07              ; every 8th frame
    jr   nz, .no_hit
    ; Set envelope period (controls decay speed)
    ld   a, 11
    ld   d, $80           ; period = $0080 -- medium decay
    call ay_write_d
    ld   a, 12
    ld   d, 0
    call ay_write_d
    ; Trigger: write shape $00 (single decay)
    ld   a, 13
    ld   d, $00
    call ay_write_d
    ; Volume C = envelope mode
    ld   a, 10
    ld   d, $10
    call ay_write_d
.no_hit:
```

This saves several bytes by eliminating the manual decay code. The trade-off: the envelope generator is shared across all channels in envelope mode. If channel A is using E+T drone, channel C cannot independently use the envelope for drum decay. Plan your channel allocation accordingly.

---

5. Multi-Channel Harmony

The AY has three independent tone channels. AY-beat can derive all three from a single formula using bit rotation, creating the impression of counterpoint from almost no code.

Three Voices from One Formula

```
; Three related voices from one frame counter
    ld   a, (frame_counter)
    ld   e, a

    ; === Channel A: base formula ===
    and  $3F              ; period 0-63
    ld   d, a
    xor  a                ; R0
    call ay_write_d

    ; === Channel B: same formula, rotated 2 bits ===
```

```
    ld   a, e
    rrca
    rrca                ; rotate right by 2
    and  $3F
    ld   d, a
    ld   a, 2           ; R2
    call ay_write_d

    ; === Channel C: same formula, rotated 4 bits ===
    ld   a, e
    rrca
    rrca
    rrca
    rrca                ; rotate right by 4
    and  $3F
    ld   d, a
    ld   a, 4           ; R4
    call ay_write_d
```

The bit rotations create phase-shifted versions of the same pattern. The channels play related but offset melodies – they follow the same contour but arrive at each pitch at different times. This creates an impression of counterpoint: multiple independent voices that share an underlying logic.

Why Rotation Creates Harmony

RRCA is a *rotation*, not a shift – bits that fall off the bottom wrap to the top. This means the three channels cycle through the same set of period values in the same order, but offset in time. The offset depends on the rotation amount:

- **RRCA x 2:** The channel is "ahead" by approximately a quarter of the pattern cycle. This often creates intervals that sound like fourths or fifths – not precisely tuned, but harmonically related enough to be pleasant.
- **RRCA x 4:** Half a byte's worth of offset. This tends to produce octave-like relationships, since bit 4 being rotated to bit 0 effectively halves the period in certain phase alignments.

These are not real musical intervals. They are pseudo-harmonic relationships created by the structure of binary numbers. But the ear is forgiving – if two tones share most of their bit pattern, they sound "related," and that is enough for a 256-byte intro.

Volume Formulas for Multi-Channel

Give each channel a different volume formula to avoid all three voices being at the same level simultaneously:

```
    ; Volume A: bits 6-3 of t
    ld   a, e
    rrca
    rrca
    rrca
    and  $0F
    ld   d, a
```

```
    ld   a, 8
    call ay_write_d

    ; Volume B: bits 4-1 of t (different phase)
    ld   a, e
    rrca
    and  $0F
    ld   d, a
    ld   a, 9
    call ay_write_d

    ; Volume C: inverted bits 5-2 of t
    ld   a, e
    rrca
    rrca
    and  $0F
    xor  $0F              ; invert -- when A is loud, C is quiet
    ld   d, a
    ld   a, 10
    call ay_write_d
```

The inverted volume on channel C creates a call-and-response dynamic: as one voice fades in, another fades out. This costs 2 extra bytes (the `XOR $0F`) but significantly improves the musical texture.

---

6. Formula Cookbook

The following formulas have been tested on the AY at 50 Hz frame rate. “Bytes” refers to the Z80 implementation cost for computing the formula from a value already in register A (the frame counter). The period mask determines the pitch range.

Tone Period Formulas

#	Formula	Z80 Implementation	Bytes	Sound	Best For
1	`t AND $3F`	`and $3F`	2	Rising saw-tooth, 1.28-sec cycle	Simple sweep
2	`t*3 AND $3F`	`ld e,a : add a,a : add a,e : and $3F`	5	Faster sweep, wider intervals	Energetic bass
3	`t XOR (t>>3)`	`ld e,a : rrca : rrca : rrca : xor e`	5	Chaotic with periodic struc-ture	Noise texture

#	Formula	Z80 Implementation	Bytes	Sound	Best For
4	`(t AND $0F) XOR $0F`	`and $0F : xor $0F`	4	Triangle wave, ping-pong sweep	Melodic lead
5	`t*5 AND t>>2`	`ld e,a : add a,a : add a,a : add a,e : ld d,a : ld a,e : rrca : rrca : and d`	10	Rhythmic gating	Percussion-like
6	`(t+t>>4) AND $1F`	`ld e,a : rrca : rrca : rrca : rrca : add a,e : and $1F`	6	Slowly modulated sweep	Evolving drone
7	`t AND (t>>3) AND $1F`	`ld e,a : rrca : rrca : rrca : and e : and $1F`	6	Self-similar, fractal rhythm	Complex patterns
8	`(t>>1) XOR (t>>3)`	`ld e,a : rrca : ld d,a : rrca : rrca : xor d`	6	Dual-speed interference	Metallic texture
9	`t*7 AND $7F`	`ld e,a : add a,a : add a,a : add a,a : sub e : and $7F`	6	Wide sweep, 7x speed	Fast arpeggio feel
10	`(t XOR t>>1) AND $3F`	`ld e,a : rrca : xor e : and $3F`	5	Gray code sequence	Staircase melody
11	`t AND $07 OR t>>4`	`ld e,a : and $07 : ld d,a : ld a,e : rrca : rrca : rrca : rrca : or d`	8	Nested loops, two rhythmic layers	Layered rhythm
12	`(t+t+t>>2) AND $3F`	`ld e,a : add a,a : ld d,a : ld a,e : rrca : rrca : add a,d : and $3F`	8	Accelerating sweep with sub-pattern	Textured lead

Volume Formulas

#	Formula	Z80	Bytes	Effect
V1	`t>>3 AND $0F`	`rrca : rrca : rrca : and $0F`	5	Slow fade cycle, 5.12 sec
V2	`(t AND $0F) XOR $0F`	`and $0F : xor $0F`	4	Triangle volume, ping-pong

#	Formula	Z80	Bytes	Effect
V3	`t*3>>4 AND $0F`	`ld e,a : add a,a : add a,e : rrca : rrca : rrca : rrca : and $0F`	8	Irregular fade pattern
V4	`$0F` (constant)	`ld d,$0F`	2	Maximum volume, use with envelope mode

How to Read the Table

Pick a tone formula and a volume formula. Combine them. The total byte cost is the sum of both implementations plus the AY register write overhead (~8 bytes per channel for two ay_write calls: register select + data for tone low and volume). A single channel with formula #1 and volume V1 costs approximately 2 + 5 + 16 = 23 bytes including register writes.

Formula #10 (Gray code) deserves special mention. The Gray code sequence only changes one bit per step, so the tone period changes by exactly one unit per frame – a smooth, staircase-like melody. Combined with the AND mask, it cycles through a limited pitch range with pleasant regularity. This is one of the most musical-sounding single formulas.

---

7. Putting It Together: A Complete AY-Beat Engine

Here is a complete, minimal AY-beat engine that produces 3-channel generative sound with envelope drone. This is the engine you drop into a 256-byte intro alongside your visual effect.

```
; ============================================================
; Complete AY-beat engine -- 47 bytes
; Produces 3-channel generative music with envelope drone
; Call once per frame (after HALT)
; Clobbers: AF, BC, DE, HL
; ============================================================

ay_beat:
    ld   hl, .frame
    ld   a, (hl)          ; A = frame counter
    inc  (hl)             ; advance for next frame
    ld   e, a             ; E = t (preserved copy)

    ; --- Mixer: all three tones on, no noise ---
    ; Only needed on first frame, but costs 5 bytes either way
    push af
    ld   a, 7             ; R7
```

```
    ld   d, $38           ; tones A+B+C on, noise off
    call .wr
    pop  af

    ; --- Channel A: tone = t AND $3F ---
    and  $3F
    ld   d, a
    xor  a                ; R0
    call .wr

    ; --- Channel B: tone = t*3 AND $3F ---
    ld   a, e
    add  a, a
    add  a, e             ; A = t * 3
    and  $3F
    ld   d, a
    ld   a, 2             ; R2
    call .wr

    ; --- Channel C: tone = (t XOR t>>1) AND $3F ---
    ld   a, e
    rrca
    xor  e
    and  $3F
    ld   d, a
    ld   a, 4             ; R4
    call .wr

    ; --- Volumes: A and B fixed at 12, C = envelope mode ---
    ld   a, 8             ; R8 = Volume A
    ld   d, 12
    call .wr
    ld   a, 9             ; R9 = Volume B
    ld   d, 10
    call .wr
    ld   a, 10            ; R10 = Volume C
    ld   d, $10           ; envelope mode
    call .wr

    ; --- Envelope: period sweeps with t, triangle shape ---
    ld   a, e
    rrca
    rrca
    and  $3F
    add  a, $20           ; keep envelope period above $20
    ld   d, a
    ld   a, 11            ; R11
    call .wr
    ; R12 = 0 (envelope period high)
    inc  a                ; A = 12
    ld   d, 0
    call .wr
```

```
    ; Shape: only write on frame 0 to avoid constant restarts
    ld   a, e
    or   a
    ret  nz               ; skip shape write on all frames except 0
    ld   a, 13            ; R13
    ld   d, $0E           ; shape $0E = repeating triangle /\/\
    ; fall through to .wr, then ret

.wr:
    ; Write D to AY register A
    ld   bc, $FFFD
    out  (c), a           ; select register
    ld   b, $BF
    out  (c), d           ; write value
    ret

.frame:
    DB   0                ; frame counter (self-modifying data)

; ============================================================
; Total: 47 bytes (code) + AY write routine shared
; The .wr routine is 9 bytes. If your intro already has an
; AY write routine, the engine body alone is 38 bytes.
; ============================================================
```

What This Produces

- **Channel A:** A simple ascending sweep, cycling through periods 0-63 every 64 frames (1.28 seconds). The fundamental pattern.
- **Channel B:** The same sweep at 3x speed, creating faster-moving intervals. When it aligns with channel A, you hear consonance; when it diverges, you hear dissonance. The alternation creates rhythmic interest.
- **Channel C:** A Gray code sweep in envelope mode. The triangle envelope creates automatic volume modulation, producing a drone that phases against the tone period. This is the harmonic bed underlying the other two voices.
- **Overall:** An evolving, self-similar texture that cycles through tonal relationships. It sounds alien and mechanical – exactly right for a 256-byte intro.

Customisation Points

Change the tone formulas. Swap any of the AND/RRCA sequences for a different formula from the cookbook (section 6). Each substitution changes the character entirely.

Add noise percussion. Insert a `ld a,e : and $07 : jr nz,.no_hit` block (section 4) to add rhythmic kicks. Cost: ~12 bytes. Steal a channel (typically B) or overlay noise on channel C.

Use pentatonic masking. Instead of `AND $3F` as the final mask, index into a 5-byte pentatonic lookup table. This constrains the tone periods to harmonically related values, making the output sound more deliberately musical. Cost: ~8 bytes (5 for the table, 3 for the lookup). Chapter 13 discusses this technique.

Vary fixed volumes. Replace the constant volume writes with volume formulas from section 6. Even `ld a,e : rrca : rrca : rrca : and $0F` (5 bytes per channel) adds significant dynamic interest.

---

8. Advanced: Combining Techniques

The preceding sections cover individual building blocks. A well-crafted AY-beat engine combines several:

Architecture for a 256-Byte Intro

```
Frame 0:   Set mixer, envelope shape (one-time setup)
Frame N:   Update tone A (melody formula)
           Update tone B (harmony formula, rotated)
           Update volume A (fade formula)
           Update volume B (inverted fade)
           Channel C in E+T drone mode (auto-evolving)
           Every 8th frame: noise hit on C (toggle mixer)
```

The total CPU cost per frame is approximately 300-500 T-states – well under 1% of the ~70,000 T-states available per frame. The remaining 99% is available for your visual effect.

Register Budget

The AY has 14 writable registers. In a minimal AY-beat engine, you typically write 8-10 per frame:

Register	Written	Source
R0 (Tone A low)	Every frame	Formula
R2 (Tone B low)	Every frame	Formula
R4 (Tone C low)	Every frame or once	Formula or fixed
R1, R3, R5 (Tone high)	Once (set to 0)	Constant
R7 (Mixer)	Every frame or once	Constant or toggled for noise
R8, R9 (Volume A, B)	Every frame	Formula or constant
R10 (Volume C)	Once	$10 (envelope mode)
R11 (Envelope low)	Every frame	Formula
R13 (Envelope shape)	Once (frame 0)	Constant

Registers you can skip entirely: R6 (noise period – only needed if using noise), R12 (envelope high – set once to 0 for short periods), R14-R15 (I/O ports – irrelevant for sound).

Size Breakdown

For a 256-byte intro, every byte matters. Here is how a typical AY-beat budget looks:

Component	Bytes
AY write routine	9
Frame counter management	5
3 tone formulas (simple)	12-18
3 volume settings	6-15
Mixer setup	5
Envelope setup	8-12
Total	**45-64**

This leaves 192-211 bytes for the visual effect, the main loop, and any other infrastructure. At 45 bytes, the engine in section 7 is close to optimal for the amount of sound it produces.

---

9. AY-as-DAC: Classic Bytebeat Through the Volume Register

There is a middle path between the beeper dead end and the AY-beat reimagining. The AY-3-8910's volume registers (registers 8, 9, 10) accept 4-bit values (0-15). If you update a volume register at a high rate – say, during a tight loop – the AY output becomes a 4-bit DAC. This is how digitised speech and sample playback work in Spectrum demos.

Applied to bytebeat: compute `f(t)`, shift right to 4 bits, write to volume register:

```
; AY-as-DAC bytebeat -- 4-bit PCM through volume register
; Still costly (~80% CPU), but sounds better than beeper
    ld   a, 7               ; mixer: all channels off (tone+noise)
    ld   d, %00111111
    call ay_write
    ld   de, 0               ; t = 0
.loop:
    ; Compute f(t): t AND (t >> 5) -- classic bytebeat formula
    ld   a, e
    ld   b, d
    srl  b
    rr   a
    srl  b
    rr   a
    srl  b
    rr   a
    srl  b
    rr   a
    srl  b
    rr   a                   ; A = t >> 5 (using DE as 16-bit t)
    and  e                   ; A = t AND (t >> 5)
    rrca
    rrca
    rrca
    rrca
```

```
    and  $0F                ; scale to 4-bit (0-15)
    ld   bc, AY_REG
    ld   b, $FF             ; select register
    push af
    ld   a, 8               ; register 8: volume A
    out  (c), a
    ld   b, $BF
    pop  af
    out  (c), a             ; write volume
    inc  de                 ; t++
    jr   .loop
```

This produces recognisable bytebeat – the actual waveform formulas from section 1, audible through the AY. The sound quality is better than the beeper (4-bit resolution vs 1-bit), and the AY's output stage provides proper audio levels.

The cost is still brutal: ~80% CPU. You get a thin sliver of time for visuals – enough for a slowly-updating attribute effect, not enough for anything ambitious. This technique is useful when you want the *specific sound* of classic bytebeat formulas and are willing to pay the CPU price.

Three Output Paths Compared

Path	Resolution	CPU cost	Sound character	Practical for demos?
Beeper (port $FE)	1-bit	~100%	Harsh, buzzy	No
AY volume DAC	4-bit	~80%	Classic bytebeat	Barely (attribute effects only)
AY-beat (registers)	Tone/noise	~0.5%	Chip music, generative	Yes – the right choice

For size-coded intros and demos, AY-beat is almost always the correct choice. Reserve AY-as-DAC for art projects where the specific bytebeat sound aesthetic is the point.

---

10. Music Theory for Algorithms

AY-beat formulas that ignore music theory produce interesting noise. Formulas that *encode* music theory produce actual music. The following techniques add musicality for minimal bytes.

Scale Tables: Constraining Output to Pleasant Notes

A raw formula like `tone = t AND $3F` produces all 64 possible period values – most of which are not musically useful. A **scale table** maps formula output to actual

note periods, ensuring every value sounds good.

Scale	Notes	Table size	Character
Pentatonic	5 (C D E G A)	10 bytes (5 × 2-byte periods)	Always consonant, folk/world feel
Diatonic major	7 (C D E F G A B)	14 bytes	Bright, Western, familiar
Diatonic minor	7 (C D Eb F G Ab Bb)	14 bytes	Dark, melancholic
Blues	6 (C Eb F F# G Bb)	12 bytes	Gritty, expressive
Chromatic	12	24 bytes	Atonal, dissonant – usually wrong for sizecoding

The pentatonic scale is the size-coder's best friend: 5 notes, 10 bytes, and *any* combination of notes sounds acceptable. You cannot play a wrong note on a pentatonic scale. This is why so many 256-byte intros sound vaguely "Asian" or "folk" – the pentatonic constraint makes random sequences musical.

```
; Scale-constrained note lookup
; Input: A = formula output (any value)
; Output: DE = AY tone period
    ; Map to scale index: A mod scale_length
    and  $07               ; keep low 3 bits
    cp   5                 ; pentatonic has 5 notes
    jr   c, .in_range
    sub  5                 ; wrap: 5→0, 6→1, 7→2
.in_range:
    add  a, a              ; ×2 for word entries
    ld   hl, pentatonic
    add  a, l
    ld   l, a
    ld   e, (hl)
    inc  hl
    ld   d, (hl)           ; DE = tone period
```

Octave Derivation: Free Pitch Range

Store one octave of periods. Derive all others by bit-shifting:

- `SRL D : RR E` = one octave up (period halved, pitch doubled)
- `SLA E : RL D` = one octave down (period doubled, pitch halved)

Five pentatonic notes × one stored octave × bit-shifting = 5 notes × 5+ octaves = 25+ distinct pitches from 10 bytes of data. The formula selects the note, a separate bit mask selects the octave:

```
    ; note_index = formula AND $0F
    ; octave = note_index / 5 (0-2)
    ; note = note_index % 5
    ; Look up base period, then SRL 'octave' times
```

Arpeggio: Chord Tones in Sequence

An arpeggio cycles through the tones of a chord. In scale-degree terms:

Chord	Scale offsets	Sound
Major triad	0, 2, 4 (root, third, fifth)	Bright, resolved
Minor triad	0, 2, 3 (root, m.third, fifth)	Dark, tense
Power chord	0, 4 (root, fifth)	Open, strong
Suspended	0, 3, 4 (root, fourth, fifth)	Ambiguous, floating

Implementation: `arp_step = (t / speed) % chord_size`, then add the offset to the current root note:

```
; Arpeggio: cycle through major triad
    ld   a, (frame)
    rrca
    rrca                    ; A = frame / 4 (arp speed)
    ; mod 3 for three chord tones
    ld   b, a
.mod3:
    sub  3
    jr   nc, .mod3
    add  a, 3               ; A = 0, 1, or 2
    ld   hl, arp_major
    add  a, l
    ld   l, a
    ld   a, (hl)            ; A = scale offset
    ; add to chord root, look up in scale table
    ; ...

arp_major:  DB  0, 2, 4     ; root, third, fifth (3 bytes)
arp_minor:  DB  0, 2, 3     ; root, min.third, fifth (3 bytes)
```

Three bytes per chord shape. The arpeggio speed is derived from the frame counter — no separate timer needed.

Step Ornaments: Trills, Mordents, and Slides

An ornament is a tiny cyclic pattern of relative pitch offsets applied to a note. In tracker music, ornaments make flat tones come alive:

Ornament	Pattern	Effect	Bytes
Trill	0, +1, 0, -1	Rapid alternation with neighbor	4
Mordent	0, +1, 0, 0	Brief upper neighbor, then settle	4
Slide up	0, 0, +1, +1	Gradual rise	4
Vibrato	0, +1, +1, 0, -1, -1	Smooth wobble	6

Apply by adding the ornament value to the note index before the scale table lookup:

```
    ; ornament_pos = (frame) AND (ornament_length - 1)
    ld   a, (frame)
```

```
    and  $03              ; mod 4 for 4-step ornament
    ld   hl, trill
    add  a, l
    ld   l, a
    ld   a, (hl)          ; A = pitch offset (-1, 0, or +1)
    add  a, c             ; C = current note index
    ; ... look up modified note in scale table

trill:    DB  0, 1, 0, -1   ; 4 bytes
mordent:  DB  0, 1, 0, 0    ; 4 bytes
```

Four bytes transform a static tone into a living voice. Stack multiple ornaments on different channels for rich texture.

Chord Progressions: Harmonic Movement

The chord root can change over time, following a progression. Classical harmony in 4 bytes:

```
; I - IV - V - I progression (the backbone of Western music)
progression:  DB  0, 3, 4, 0     ; scale degrees

; Select chord: (frame / 64) AND 3
    ld   a, (frame)
    rrca
    rrca
    rrca
    rrca
    rrca
    rrca                  ; A = frame / 64
    and  $03              ; mod 4
    ld   hl, progression
    add  a, l
    ld   l, a
    ld   a, (hl)          ; A = chord root (scale degree)
```

Four bytes of progression data, cycled by the frame counter, give your AY-beat piece harmonic movement – the sense that it is “going somewhere” rather than looping on one chord. Other progressions:

Progression	Degrees	Bytes	Feel
I-IV-V-I	0, 3, 4, 0	4	Classic resolution
I-V-vi-IV	0, 4, 5, 3	4	Pop/rock standard
i-VI-III-VII	0, 5, 2, 6	4	Epic minor
I-I-I-I	0, 0, 0, 0	1 (or skip)	Drone/meditative

Total Data Budget for Rich Music

Combining all techniques:

Component	Bytes
Pentatonic table (5 notes)	10
Arpeggio pattern (1 chord)	3
Ornament (trill)	4
Progression (4 chords)	4
Total	**21**

21 bytes of musical data — plus ~45 bytes of engine code — produces three-channel music with melody, harmony, chord changes, and ornamentation. The `aybeat.a80` example in this book's companion code demonstrates this approach in 320 bytes, with room left over for visuals.

---

11. L-System Grammars: Fractal Melodies

Lindenmayer systems (L-systems) are rewriting grammars originally invented to model plant growth. Applied to music, they generate self-similar sequences with long-range structure from tiny rule sets.

The Concept

An L-system has an **axiom** (starting string) and **production rules** (expansion rules). Each iteration replaces every symbol according to its rule:

```
Axiom: A
Rules: A → A B,  B → A

Step 0: A
Step 1: A B
Step 2: A B A
Step 3: A B A A B
Step 4: A B A A B A B A
```

This is the **Fibonacci L-system**. The sequence grows by the Fibonacci ratio (~1.618x per step). Map the symbols to musical events:

Symbol	Musical meaning
A	Play root note (scale degree 0)
B	Play fifth (scale degree 4)

The resulting melody: root, fifth, root, root, fifth, root, fifth, root... — a sequence that is neither periodic nor random, but *quasi-periodic*. It has structure at every scale, like a fractal. It sounds intentional without being repetitive.

Why L-Systems Work for Music

1. **Self-similarity.** The melody at large scales echoes the melody at small scales. This is what makes composed music feel coherent – themes recur at different levels.

2. **Non-repetition.** Unlike a looped pattern, an L-system sequence never exactly repeats (for irrational growth ratios). It stays interesting.
3. **Tiny encoding.** The rules are a few bytes. The sequence they generate is arbitrarily long.

Useful L-System Rules

Name	Axiom	Rules	Growth	Character
Fibonacci	A	A→AB, B→A	~1.618x	Quasi-periodic, organic
Thue-Morse	A	A→AB, B→BA	2x	Balanced, fair — no long runs
Period-doubling	A	A→AB, B→AA	2x	Increasingly syncopated
Cantor	A	A→ABA, B→BBB	3x	Sparse, with silences (B=rest)

Z80 Implementation

The trick for Z80 is to **not expand the string in memory** (that would require unbounded buffer space). Instead, compute the symbol at position `n` recursively: trace back through the rule applications to determine which original symbol position `n` came from.

For the Fibonacci L-system, there is an elegant shortcut. The symbol at position `n` depends on the Zeckendorf representation (Fibonacci coding) of `n`. But for practical sizecoding, a simpler approach works:

```
; L-system melody generator (Fibonacci: A→AB, B→A)
; Returns next note in sequence
; Uses position counter in memory
;
; The sequence of symbols can be generated iteratively:
; keep two "previous" bytes and generate the next

lsys_next:
    ld   hl, lsys_state
    ld   a, (hl)           ; prev1
    inc  hl
    ld   b, (hl)           ; prev2
    inc  hl
    ld   c, (hl)           ; position in current generation

    ; Fibonacci rule: output prev1, then swap
    ; When position reaches length, expand to next generation
    ld   d, a              ; D = current symbol to output

    ; Advance: shift the pair
    inc  c
    ld   (hl), c
    dec  hl
    ld   (hl), a           ; prev2 = prev1
    dec  hl
    ; New prev1 from rule: A→A (first output), then A→B (second)
    ; Simplified: alternate symbols based on parity
```

```
    ld   a, c
    and  $01
    jr   z, .sym_a
    ld   a, 1              ; B
    jr   .store
.sym_a:
    xor  a                 ; A (=0)
.store:
    ld   (hl), a

    ; Map symbol to scale degree
    ld   a, d
    or   a
    jr   z, .root
    ; B = fifth
    ld   a, 4              ; scale degree 4 = fifth in pentatonic
    ret
.root:
    xor  a                 ; scale degree 0 = root
    ret

lsys_state:
    DB   0                 ; prev1 (A=0, B=1)
    DB   0                 ; prev2
    DB   0                 ; position
```

A more practical approach for sizecoding: precompute several iterations of the L-system into a short buffer at init time (one iteration of Fibonacci from a 5-symbol axiom yields 8 symbols, two iterations yield 13, three yield 21 — all fitting in a small buffer), then loop through the buffer as a melody sequence:

```
; Precompute L-system into buffer (Fibonacci, 3 iterations)
; Axiom: "AABAB" (5 symbols) → 8 → 13 → 21 symbols
; 21 notes of fractal melody from 5 bytes of axiom + expansion code

lsys_expand:
    ld   hl, lsys_axiom
    ld   de, lsys_buf
    ld   b, 5              ; axiom length
.expand_iter:
    ; One iteration: for each symbol, apply rule
    push bc
    push hl
    ld   hl, lsys_buf
    ld   de, lsys_work     ; expand into work buffer
    ; ...expand according to rules...
    pop  hl
    pop  bc
    ; Copy work back to buf for next iteration
    ; Repeat for desired number of iterations
    ret

lsys_axiom:
```

```
    DB   0, 0, 1, 0, 1     ; A A B A B

; During playback:
; melody_index = frame / note_duration
; note = lsys_buf[melody_index % buf_length]
; look up in scale table → AY period
```

Melody as Motion, Not Absolute Notes

The most musical use of L-systems is not mapping symbols to fixed notes, but mapping them to **scale step directions**. A melody is fundamentally about *motion* — up, down, repeat, skip — on a scale. The starting note is arbitrary; the contour is what matters.

Define symbols as movements:

Symbol	Meaning	Scale step
U	Step up	+1
D	Step down	-1
R	Repeat	0
S	Skip up (leap)	+2

Now an L-system generates melodic *contour*, not fixed pitch sequences:

```
Axiom: U
Rules: U → U R D,  D → U,  R → U D

Step 0: U                          (+1)
Step 1: U R D                      (+1, 0, -1)
Step 2: U R D  U D  U              (+1, 0, -1, +1, -1, +1)
Step 3: U R D  U D  U  U R D  U  U R D  U D   ...
```

The melody walks up and down the current scale, always staying within the scale table. It naturally tends toward the starting pitch (the returns balance the departures), creating the tension-and-resolution arc that makes music feel intentional.

```
; Motion-based L-system playback
; current_note = scale index, modified by each symbol
    ld   a, (current_note)
    ld   hl, lsys_buf
    ld   b, (melody_pos)
    add  a, l
    ; ... get motion symbol at current position ...
    ; D = motion offset from symbol table
    add  a, d              ; current_note += motion
    and  $0F               ; wrap to scale range
    ld   (current_note), a
    ; look up in pentatonic table → AY period
```

This is more musical than mapping A=root, B=fifth. The same L-system rules produce different melodies depending on the starting note and the underlying scale — change the scale from pentatonic to blues and the same contour produces a completely different mood.

Tribonacci: Three Symbols for Richer Patterns

The Fibonacci L-system uses two symbols. **Tribonacci** uses three: A→ABC, B→A, C→B. The growth ratio is ~1.839x (the tribonacci constant). Three symbols mean more varied melodic content:

Symbol	As motion	As note
A	Step up (+1)	Root
B	Repeat (0)	Third
C	Step down (-1)	Fifth

```
Axiom: A
Step 1: A B C
Step 2: A B C  A  B
Step 3: A B C  A  B  A B C  A B C
```

The tribonacci sequence has longer non-repeating runs than Fibonacci and a more complex internal structure. Musically, the three-symbol vocabulary gives melodies more variety — they don't just ping-pong between two states.

PRNG Melodies with Curated Seeds

A linear feedback shift register (LFSR) or similar PRNG generates a deterministic pseudo-random sequence from a seed value. The sequence *sounds* random but repeats exactly if you reset the seed. This gives you reproducible melody fragments.

The technique: **pre-test many seeds, keep the ones that sound good.** Store 2-4 seed values (2 bytes each) for different sections of your piece. At runtime, load the seed and let the PRNG generate the melody. The PRNG itself is ~6-8 bytes; each seed is 2 bytes.

```
; LFSR-based melody generator
; HL = seed (determines the melody)
prng_note:
    ld   a, h
    xor  l              ; mix bits
    rrca
    rrca
    xor  h
    ld   h, a
    ld   a, l
    add  a, h
    ld   l, a           ; advance LFSR state (~6 bytes)
    and  $07            ; constrain to scale range
    ret                 ; A = note index for scale table

; Different seeds → different melodies
seed_verse:   DW  $A73B    ; tested: produces ascending contour
seed_chorus:  DW  $1F4D    ; tested: produces energetic pattern
seed_bridge:  DW  $8E21    ; tested: produces descending, calm
```

The workflow: write a test harness that plays the PRNG melody for each seed value 0-65535, listen (or analyse), mark the good ones. In practice, a few hours of testing

yields dozens of usable seeds. Store 3-4 of them and switch between sections of your piece.

Combining with scale tables: the PRNG output feeds through the pentatonic table, so even "bad" seeds produce consonant notes. You're curating for *melodic contour*, not avoiding wrong notes — the scale table already handles that.

Combining with L-systems: use the PRNG to *select which L-system rule to apply* at each step, creating stochastic L-systems. The seed controls the "personality" of the piece; the grammar rules control the structure. This hybrid produces the richest output from the fewest bytes.

Combining L-Systems with Other Techniques

L-systems generate note *sequences*. Combine with the other techniques from this appendix:

- **Scale table** maps L-system symbols to actual AY periods
- **Ornaments** add expression to each note
- **Arpeggio** turns each L-system note into a chord
- **Envelope drone** provides a sustained harmonic bed under the fractal melody
- **Chord progression** changes the root — the L-system melody is transposed to each chord

The result: a tiny program (~60-80 bytes of music code + 20 bytes of data) generating minutes of structurally coherent, non-repeating, harmonically grounded music. This is algorithmic composition, not random noise — and it fits in a size-coded intro.

Other Grammars for Music

Beyond L-systems, other formal grammars produce interesting musical sequences:

Cellular automata. Rule 30 or Rule 110, applied to a row of bits, produce complex patterns. Map bit positions to note on/off events. Cost: ~15 bytes for the CA rule, ~20 bytes for the stepper.

Euclidean rhythms. Distribute `k` beats evenly across `n` steps. This algorithm (related to the Euclidean GCD) generates rhythmic patterns found in music worldwide: 3-in-8 is tresillo, 5-in-8 is cinquillo, 7-in-12 is a common West African bell pattern. Implementation is ~20 bytes and produces perfect rhythmic foundations for any AY-beat engine.

See Also

- **Chapter 11** – AY-3-8910 architecture, tone/noise/envelope theory, buzz-bass technique
- **Chapter 12** – Music engine integration, sync to effects, hybrid digital drums
- **Chapter 13** – Sizecoding techniques, where AY-beat fits in the 256b/512b/1K/4K size tiers
- **Appendix G** – Complete AY register reference with bit layouts, port addresses, and note tables

---

Sources: Viznut (Ville-Matias Heikkila), "Algorithmic symphonies from one line of code – how and why?" (2011); countercomplex.blogspot.com; Chapter 13 of this book; various 256-byte ZX Spectrum intros from Pouet.net

What's New

v12 (2026-02-24)

Comprehensive review pass. Full technical audit of all 23 chapters with systematic fixes:

12 HIGH severity fixes: - Ch.02: RET cc T-state values corrected (5/11T → 11/5T taken/not-taken) across 6 comments + timing table - Ch.04: Patrik Rak and Raxoft identified as same person throughout - Ch.07: `SET n,(DE)` / `RES n,(DE)` rewritten — instructions don't exist on Z80, replaced with (HL) approach - Ch.10: False claim that `LD (addr),SP` doesn't exist removed (ED 73, 20T) - Ch.11: AY mixer value corrected ($28 → $18 for noise C) - Ch.15: Contended memory table corrected — only $4000-$7FFF contended on 48K, not all RAM - Ch.16: XOR sprite inner loop timing corrected (98T → 134T), JR NZ annotations fixed throughout - Ch.18: `spawn_bullet` bug fixed — D register zeroed before `ADD IX,DE` - Ch.19: AABB worst-case recounted (156T → ~270T), `tile_at` recounted (40T → ~182T) - Ch.23: DOWN_HL argument rewritten — was claiming correct code was buggy

19 MEDIUM severity fixes: LDIR formula standardised to (N-1)×21+16, PUSH operation order corrected, byte counts fixed, Agon 60Hz corrected, IM2 IY save added, invalid `ld hl,a` replaced, and more.

AI smell cleanup: Removed chain-of-thought leaks ("Wait – that is wrong"), cut philosophical padding sections, eliminated word-level tics (remarkably, deceptively simple, paradigm shift, borders on magic, etc.), trimmed defensive balancing and motivational pep talks across 9 chapters.

Two new tools: - `tools/autotag.py` — semi-automatic code block classifier and tagger (–preview, –apply, –stats) - `tools/audit_tstates.py` — T-state audit comparing inline annotations with computed values (–scan-chapters, –asm-check)

Code block pipeline: - 79 bare code blocks classified and tagged with language (z80/mermaid/text) - 279 code blocks tagged with `id:chNN_slug` identifiers - 271 .z80 + 8 .mmd canonical listings extracted to `listings/` - Final audit: 0 WRONG, 0 PARTIAL T-state annotations - All 29 assembly units compile with sjasmplus

v11 (2026-02-24)

- Art-top/Gogin attribution fix across affected chapters.
- MCC sidebar content added.
- ZXDN research integrated.

v10 (2026-02-24)

- External listings system (`tools/manage_listings.py`): extract, inject, verify, stats commands.
- Unified versioning via `version.json` + `build_book.py`.
- UNDER CONSTRUCTION banner for work-in-progress chapters.

v0.8 (2026-02-24)

Chapters expanded. Three thin chapters significantly deepened:

- **Chapter 7 (Rotozoomer):** Three rotozoomer variants compared – monochrome bitmap (full pixel), chunky (2x2/4x4), and attribute-based (32x24 colour grid). New sections on fixed-point stepping mechanics, self-modifying code at the byte level, texture design, and boundary handling. Credits corrected: GOA4K and Refresh are by Exploder^XTM.
- **Chapter 10 (Dotfield Scroller):** POP-trick deep dive with interrupt risks and SP-save patterns. Lissajous, helix, and multi-wave trajectory formulas. 4-phase colour cycling code with attribute byte manipulation. Demoscene lineage from Born Dead to Introspec's Eager.
- **Chapter 13 (Size-Coding):** Now covers the full range from 256 bytes to 4K intros. New size-coder's toolkit (register assumptions, DJNZ tricks, RST as 1-byte CALL, overlapping instructions). 4K intro section with GOA4K (Exploder^XTM), inal and Megademica (SerzhSoft). AY bytebeat and procedural graphics compo sections added.

Five new appendices (all 9 now complete, A–I):

- **E: eZ80 Quick Reference** – ADL mode, MLT/LEA/PEA/TST, Agon Light 2 specifics.
- **F: Z80 Variants** – Z80N (Next) instructions organised by the pain they solve, R800 (MSX turboR) pipeline and MULUB/MULUW, Soviet clones, comparison table.
- **H: Storage APIs** – TR-DOS (Beta Disk 128) and esxDOS (DivMMC) with complete port maps, ROM API, and working code examples.
- **I: Bytebeat & AY-Beat** – Classic PCM bytebeat adapted for the AY-3-8910. Formula cookbook (12 tone + 4 volume formulas), envelope drone, music theory for algorithms (scale tables, arpeggios, ornaments, chord progressions, L-system grammars, PRNG with curated seeds).

New assembly example: `aybeat.a80` – 320-byte AY-beat engine demonstrating pentatonic scales, I-IV-V-I chord progression, arpeggio, step ornaments, envelope drone, and noise percussion. 19 bytes of musical data, three channels of generative sound.

9 new diagrams added across chapters 3, 8, 9, 14, 19, 20, 21, 22, 23. All 23 figures tagged with `<!-- figure: chNN_name -->` for greppability and translation tracking.

29 compilable examples, all passing. ~180K words (English).

v0.7 (2026-02-24)

- Appendix D: Development environment setup (sjasmplus, VS Code, DeZog, emulators).
- Three new chapter examples: ch21 game skeleton (128K, 438 lines), ch22 Agon entity system (240 lines), ch23 AI-assisted diagonal fill (149 lines).
- README rewritten with honest platform positioning ("this book lives on the ZX Spectrum"), per-chapter platform tags, TL;DR with direct download link.
- 28 compilable examples, all passing.

v0.6 (2026-02-23)

- All 23 chapters drafted. All 4 language editions (EN, ES, RU, UK) built and released.
- Translation manifest system for staleness tracking.
- ~128K words (English), ~575K words total across all languages.