

Программируя невозможное

Демосценовые техники Z80 для современных разработчиков

Alice Vinogradova

Alice Vinogradova — v21 (2026-03-01)

Содержание

Глава 1: Мыслить тактами	2
T-state: Валюта Z80	2
Машинные циклы и доступ к памяти	3
Кадр: Твой холст	4
Пентагон против «ждущих» машин	5
Оригинальные машины Sinclair	5
Пентагон: Чистый тайминг	6
Мыслить бюджетами	6
Практика: Настройка среды разработки	7
Что понадобится	8
Структура проекта	8
Конфигурация сборки	8
Практика: Тестовая обвязка	9
Чтение полосы	10
Вариации	11
Что помещается в кадр?	11
Историческая заметка: Совет Dark'a	12
Вычислительная схема	12
Итого	13
Попробуй сам	14
Глава 2: Экран как головоломка	15
Карта памяти: 6 912 байт экрана	15
Чересстрочность: Где живут строки	17
Почему?	18
Битовая раскладка: Декодирование (x, y) в адрес	18
Вычисление адреса на Z80	19
DOWN_HL: Перемещение на одну пиксельную строку вниз	20
Оптимизация Introspec'a	22
Память атрибутов: 768 байт, которые изменили всё	24
Конфликт атрибутов	25
Бордюр: Больше, чем декорация	26
Эффекты в бордюре	26
Практика: Заполнение шахматным узором	27
Что попробовать	28
Навигация по экрану: Практическая сводка	29
Перемещение вправо на один байт (8 пикселей)	29
Перемещение вниз на одну пиксельную строку	29
Перемещение вниз на одну строку знакомест (8 пикселей)	29
Перемещение вверх на одну пиксельную строку	30

Вычисление адреса атрибута по адресу пикселя	31
Попробуй сам	33
Глава 3: Инструментарий демосценера	36
Развёрнутые циклы и самомодифицирующийся код	36
Стоимость цикла	36
Развёртка: размен RAM на скорость	36
Самомодифицирующийся код: секретное оружие Z80	37
Самомодифицирующиеся переменные: паттерн \$+1	38
Стек как канал данных	39
Почему PUSH — самая быстрая запись на Z80	39
Техника	39
POP как быстрое чтение	41
Где используются PUSH-трюки	42
LDI-цепочки	43
LDI vs LDIR	43
Когда LDI-цепочки блистают	43
Битовые трюки: SBC A,A и компания	44
SBC A,A как условная маска	44
ADD A,A vs SLA A	45
Генерация кода	45
Генерация кода: написание программы, которая рисует	45
Оффлайн: генерация ассемблера из языка высокого уровня	45
Рантайм: программа пишет машинный код во время выполнения	46
Стоимость генерации	46
RET-цепочки	47
Превращение стека в таблицу диспетчеризации	47
Три стратегии для списка рендеринга	48
Компромиссы	49
Врезка: «Код мёртв» (Introspec, 2015)	49
Собираем всё вместе	49
Попробуй сам	50
Глава 4: Математика, которая реально нужна	52
Умножение на Z80	52
Метод 1: Сдвиг-и-сложение от LSB	52
Метод 2: Поиск по таблице квадратов	53
Знаковое умножение	56
Дополнительный код на практике	56
Расширение знака: идиома rla / sbc a,a	56
mul_signed — знаковое умножение 8x8	56
mul_signed_c — тонкая обёртка для отсечения задних граней	58
Сравнение стоимости	58
Деление на Z80	59
Метод 1: Сдвиг-и-вычитание (восстановливающее деление)	59
Метод 2: Логарифмическое деление	60
Синус и косинус	60
Параболическая аппроксимация	60
Рисование линий по Брезенхэму	62
Классический алгоритм и модификация Хорфа	62
Матричный метод Dark'a: сетки 8x8 пикселей	62
Завершение через ловушку	63

Арифметика с фиксированной точкой	63
Формат 8.8	63
Умножение с фиксированной точкой	64
Почему фиксированная точка важна	65
Теория и практика	65
Что Dark сделал правильно	66
Случайные числа: когда таблицы не помогают	66
Трюк с регистром R	67
Четыре генератора от сообщества	67
Подход Tribonacci из Elite	70
Генератор галактик Elite: подробнее	70
Формирование случайности	71
Seeds и воспроизводимость	71
Сравнительная таблица	72
Глава 5: 3D на 3.5 МГц	75
Проблема: двенадцать умножений на вершину	75
Метод средней точки	76
Куб как основа	76
Вывод вершин усреднением	76
Построение сложных объектов	77
Виртуальный процессор	77
Архитектура	77
Исполнение	78
Написание программ	78
Зачем виртуальный процессор?	79
Вращение	80
Вращение вокруг оси Z	80
Стоимость на базисную вершину	81
Проекция	81
Параллельная проекция	81
Перспективная проекция	82
Закрашенные полигоны	83
Отсечение задних граней	83
Сортировка по Z	84
Заливка выпуклых полигонов	84
Собираем всё вместе	85
Форма объектов	86
Практика: врачающийся закрашенный объект	87
Шаг 1: Определение объекта	87
Шаг 2: Программа средней точки	88
Шаг 3: Определение граней	88
Шаг 4: Цикл кадра	88
Исторический контекст: от журнала к демо	89
Итого	90
Глава 6: Сфера — Текстурный маппинг на 3.5 МГц	92
Проблема: круглый объект на квадратном экране	92
Ключевая идея: код, который пишет код	93
Внутри дизассемблера	93
Подсчёт тактов	94
Проход генерации кода	95

Что знал Dark: Spectrum Expert и строительные блоки	96
Дебаты на Нуре: внутренние циклы vs. математика	97
Практика: упрощённая вращающаяся сфера 56x56	98
Шаг 1: Предвычисление геометрии сферы	98
Шаг 2: Построение таблиц пропусков	98
Шаг 3: Генерация рендерящего кода	98
Шаг 4: Исполнение и отображение	100
Шаг 5: Раскладка исходного изображения	100
Общий паттерн	101
Итого	101
Глава 7: Ротозумер и чанки-пиксели	103
Что на самом деле делает ротозумер	103
Пошаговое перемещение с фиксированной точкой на Z80	104
Чанки-пиксели: размен разрешения на скорость	105
Почему 2x2 — золотая середина	105
Трюк с кодировкой \$03	105
Внутренний цикл из Illusion	106
Самомодифицирующийся код на уровне байта	107
Покадровая генерация кода	107
Перенос буфера на экран	108
Глубокое погружение: чанки-пиксели 4x4 (sq, Нуре 2022)	109
Сравнение производительности	109
Исторические корни: Born Dead #05 и сценовая преемственность	110
Практика: построение простого ротозумера	110
Дизайн текстуры и обработка границ	112
Почему выравнивание по странице, почему 256 столбцов	112
Выбор размера текстуры	112
А как насчёт границ экрана?	113
Пространство решений	113
Три подхода к вращению текстуры	114
Вариант 1: Монокромный битмап (полное пиксельное разрешение)	114
Вариант 2: Чанки-ротозумер (блоки 2x2 или 4x4)	115
Вариант 3: Атрибутный ротозумер («пиксели» из блоков 8x8)	115
Сравнение	116
Ротозумер в контексте	117
Итого	117
Глава 8: Мультиколор — Преодоление атрибутной сетки	119
Точка зрения ULA	121
Прозрение LDPUSH	121
LD DE,nn / PUSH DE	122
Сколько помещается в строку развёртки?	122
Указатель стека как курсор	123
Движок GLUF: мультиколор в настоящей игре	123
Двойная буферизация	123
Двухкадровая архитектура	124
Что видят игроки	124
Ringo: мультиколор другого рода	125
Паттерн 11110000b	125
Переключение двух экранов	125
Куда уходят такты	126

Горизонтальный скроллинг	127
Традиционный мультиколор: подход на прерываниях	127
Врезка: Black Crow #05 — ранний мультиколор	129
Палитра Spectrum и мультиколор	129
Практика: мультиколорный игровой экран	130
Шаг 1: Буфер дисплейного кода	130
Шаг 2: Смена атрибутов внутри дисплейного кода	130
Шаг 3: Рендеринг тайлов через патчинг операндов	131
Шаг 4: Наложение спрайтов	131
Шаг 5: Главный цикл	132
Что значит работа DenisGrachev'a	133
Итого	133
Попробуй сам	133
Глава 9: Атрибутные туннели и хаос-зумеры	135
Атрибутная сетка как фреймбуфер	135
Плазма: цветовой движок	136
Четвёртная симметрия: разделяй и властвуй	138
Хаос-зумер	139
Генерация кода: Processing пишет Z80	140
Вопрос запилятора	141
Вкус скриптового движка	141
Making-of: хронология и вдохновение	142
Практика: построение упрощённого атрибутного туннеля	142
Шаг 1: Заполнение пиксельной памяти паттерном	143
Шаг 2: Таблица синусов	143
Шаг 3: Плазма для одной четверти	143
Шаг 4: Четырёхстороннее копирование	144
Шаг 5: Главный цикл	145
Ключевое наблюдение	145
Источники	146
Глава 10: Точечный скроллер и 4-фазная цветовая анимация	147
Часть 1: Точечный скроллер	147
Что видит зритель	147
Шрифт как текстура	148
Адресные таблицы на основе стека	148
Внутренний цикл	150
Арифметика бюджета кадра	150
Как закодирован отскок	152
За пределами простого синуса: Лиссажу, спираль и многоволновые паттерны	152
Часть 2: 4-фазная цветовая анимация	153
Проблема цвета	153
Трюк	153
Математика восприятия	154
Почему инверсия необходима	154
Практическая стоимость	155
Текстовый оверлей	156
Родословная на демосцене	156
Общий принцип: временной обман	157
Практика 1: Текстовый скроллер с подпрыгивающей точечной матрицей	158

Практика 2: 4-фазная анимация цветового цикла	159
Итого	159
Попробуй сам	160
Глава 11: Звуковая архитектура - AY, TurboSound и Triple AY	161
11.1 Карта регистров AY-3-8910	161
Полная таблица регистров	162
Тональные каналы (R0-R5): как работает высота тона	164
Генератор шума (R6)	164
R7: Микшер – самый важный регистр	164
Регистры громкости (R8-R10)	165
Генератор огибающей (R11-R13)	166
11.2 Чиптюн-техники на 3 каналах	167
Арпеджио: имитация аккордов	167
Buzz-bass: трюк с огибающей	168
Проблема выравнивания периодов	169
Натуральный строй: Таблица #5	169
Синтез ударных	173
Орнаменты: покадровая модуляция	175
11.3 TurboSound: 2 x AY	175
Выбор чипа	175
6 каналов, настоящее стерео	176
Модификация движка	176
11.4 Triple AY на ZX Spectrum Next	177
Расширенные возможности	177
9 каналов: оркестровое мышление	177
11.5 Архитектура музыкального движка	178
Цикл проигрывателя	179
Бюджет кадра	180
Форматы и трекеры	181
11.6 Система звуковых эффектов	181
Кражा каналов	183
Процедурные таблицы SFX	184
11.7 Собираем всё вместе: рабочий пример	185
Врезка: Beeper – краткая история невозможного	185
Врезка: Agon Light 2 – звуковая система VDP	186
11.8 Практические упражнения	187
Итого	188
Глава 12: Цифровые барабаны и синхронизация с музыкой	189
12.1 Цифровые барабаны на AY	189
Проблема: AY не может воспроизводить сэмплы	189
Цена: уничтожение ЦП	190
Решение n1k-o: гибридный барабан	190
Бюджет кадра: два кадра на удар	192
12.2 Асинхронная генерация кадров	194
Наивный подход не работает	194
Динамика буфера	195
12.3 Скриптовый движок	196
Зачем нужен скрипт	196
Внешний скрипт: последовательность эффектов	196
Внутренний скрипт: вариации внутри эффекта	197

kWORK: ключевая команда	197
12.4 Инновация GABBA: видеоредактор как инструмент таймлайна	198
Проблема код-ориентированной синхронизации	198
Ответ diver4d: Luma Fusion	199
Что это меняет	199
12.5 Потоки на Z80: другой путь	200
Проблема, переформулированная	200
Переключение контекста на базе IM2	200
Как это работает на практике	201
Модель потоков	201
Практические соображения	201
Обработчик прерываний дисплея	207
Диспетчер генератора эффектов	208
Наблюдения	209
12.7 Практические упражнения	210
Итого	210
Глава 13: Мастерство sizecoding	211
13.1 Что такое sizecoding?	211
Инструментарий сайзкодера на Z80	212
13.2 Анатомия 256-байтного интро: NHBF	214
Музыка	214
Визуальная часть	214
Головоломка: поиск перекрытий	214
Открытие Art-Top	215
Байтовый бюджет	215
Ключевые техники на 256 байтах	216
13.3 Знаменитые 256-байтные интро: что делало их гениальными	217
13.4 Трюк LPRINT	218
Как это работает	218
Эффект транспозиции	218
От пиратских загрузчиков к демо-арту	219
Почему это важно для sizecoding	219
13.5 512-байтные интро: пространство для дыхания	219
Что позволяет каждый размерный уровень	219
Типичные 512-байтные паттерны	220
Самомодифицирующиеся трюки	220
Трюк с ORG	221
13.6 4К-интро: мини-демо	221
Сжатие становится жизнеспособным	221
Музыка помещается	222
Многоэффектная структура	222
GOA4K, inal и Megademica	223
Компромиссы на 4К	223
Категории соревнований	224
13.7 Практика: пишем 256-байтное интро пошагово	225
Шаг 1: Неоптимизированная версия	225
Шаг 2: Замени CALL на RST	225
Шаг 3: Перекрой данные с кодом	225
Шаг 4: Используй состояние регистров	225
Шаг 5: Меньшие кодировки везде	226

Шаг 6: Точный подсчёт байтов	226
Финальный рывок	226
13.8 Sizecoding-музыка: Bytebeat на AY	227
Минимальный формульный движок AY	227
Техники для более качественного звука из формул	228
Bytebeat против секвенсированной музыки	228
13.9 Процедурная графика: компо Rendered GFX	229
Почему Spectrum интересен для этого	229
Распространённые подходы	230
Байтовый бюджет для искусства	230
13.10 Sizecoding как искусство	230
Итого	231
Попробуй сам	232
Глава 14: Сжатие — больше данных в меньшем пространстве	233
Проблема памяти	233
Сжатие как усилитель пропускной способности	234
Бенчмарк	234
Корпус	234
Результаты	235
Треугольник компромиссов	236
Как работает LZ-сжатие	238
ZX0 — выбор sizecoding-мастера	239
Почему ZX0 существует	239
Распаковщик	239
Когда использовать ZX0	240
RLE и дельта-кодирование	240
RLE: кодирование длин серий	240
Дельта-кодирование: храни то, что изменилось	242
Подготовка данных перед сжатием	243
Энтропия: теоретический минимум	243
Вторая производная: синусоидальные и квадратичные данные	243
Транспонирование: столбцовый порядок для табличных данных	244
Чередование плоскостей: маски и пиксели	244
Обнаружение паттернов: когда не сжимать	245
Практические преобразования для типичных демосценовых данных	245
Практический конвейер	246
От ассета к бинарнику	246
Интеграция в Makefile	246
Пример: загрузочный экран с ZX0	247
Выбор правильного упаковщика	247
Возрождение MegaLZ	248
Что значат числа на практике	248
Итого: шпаргалка по упаковщикам	249
Попробуй сам	249
Глава 15: Анатомия двух машин	251
15.1 ZX Spectrum 128K: карта памяти	251
Порт \$7FFD: переключение банков	252
Практическая карта памяти для игры на 128K	253
15.2 Спорная память: практическая правда	253
Что подвержено спорности	255

Насколько медленнее?	255
Практический ответ	256
15.3 Тайминг ULA	256
Структура кадра	256
Такт-карты: области кадра	257
Тайминг строки развёртки	257
Общий и практический бюджет	258
15.4 Плавающая шина, ULA-снег и баг \$7FFD	258
Плавающая шина	259
Баг чтения \$7FFD	259
ULA-снег	259
15.5 Различия клонов	259
Pentagon 128	260
Scorpion ZS-256	260
ZX Spectrum Next	261
15.6 Agon Light 2: другой зверь	262
Двухпроцессорная архитектура	262
Модель памяти eZ80: 24-битная плоская	263
ADL-режим и Z80-режим	263
MOS API: операционная система	264
VDP-команды: разговор с экраном	265
15.7 Сравнение платформ	266
15.8 Практика: утилита-инспектор памяти	267
Версия для Spectrum	267
Версия для Agon	269
Итого	270
Глава 16: Быстрые спрайты	272
Метод 1: XOR-спрайты	272
Простейший подход	272
Когда XOR работает	273
Когда XOR не справляется	274
Метод 2: Маскированные спрайты OR+AND	274
Отраслевой стандарт	274
Формат данных	274
Внутренний цикл	274
Подсчёт тактов	277
Побайтовое выравнивание и проблема сдвига	277
Метод 3: Предварительно сдвинутые спрайты	278
Компромисс память-скорость	278
Расчёт памяти	278
Практический компромисс	279
Метод 4: Стековые спрайты (метод PUSH)	279
Самый быстрый вывод на Z80	279
Внутренний цикл	280
Как заставить работать: предвычисленная цепочка SP	280
Цена	281
Когда использовать стековые спрайты	281
Метод 5: Скомпилированные спрайты	281
Спрайт — это код	281
Как это работает	282

Скомпилированный спрайт 16x16	283
Компромиссы	284
Скомпилированные спрайты с маскированием	284
Метод 6: Грязные прямоугольники	286
Проблема фона	286
Цикл сохранения/восстановления	286
Процедура сохранения/восстановления	286
Полный бюджет кадра	287
Порядок отрисовки и перекрытие	288
Оптимизация внутренних циклов	288
Устранение управления указателями	288
Agon Light 2: аппаратные VDP-спрайты	291
Перемещение спрайта	291
Ограничения по строкам развёртки	291
Компромисс	292
Практика: 8 анимированных спрайтов при 25 fps	292
Реализация на Spectrum	292
Реализация на Agon	294
Итого	294
Попробуй сам	295
Глава 17: Скроллинг	297
Бюджет	297
Вертикальный пиксельный скроллинг	298
Проблема чересстрочности	298
Алгоритм: сдвиг вверх на один пиксель	299
Анализ стоимости	299
Частичный скроллинг: практический подход	299
Скроллинг на 8 пикселей (один знакоряд)	300
Горизонтальный пиксельный скроллинг	301
Почему горизонтальный скроллинг дорог	301
Полный расчёт бюджета	303
Можно ли сделать лучше?	303
Скроллинг атрибутов (посимвольный)	303
Скроллинг атрибутов с помощью LDIR	304
Комбинированный метод: посимвольный скроллинг + пиксельное смещение	305
Как это работает	305
Простой комбинированный метод	306
Реализация: пиксельный сдвиг краевого столбца	306
Конвойер рендеринга	307
Скроллинг пиксельных данных на один столбец символов	308
Трюк с теневым экраном	309
Стратегия скроллинга с теневым экраном	310
Сравнение: методы скроллинга на ZX Spectrum	310
Скроллинг вправо (и проблема направления)	311
Agon Light 2: аппаратный скроллинг	313
Аппаратные смещения скроллинга	313
Тайлмаповый скроллинг	313
Загрузка столбцов через кольцевой буфер	314
Сравнение: Spectrum против Agon — скроллинг	314

Практика: горизонтальный скроллинг уровня	315
Версия для Spectrum: комбинированный посимвольный + пиксель- ный скроллинг	315
Версия для Agon: аппаратный тайлмаповый скроллинг	317
Вертикальный + горизонтальный: комбинированный скроллинг	318
Советы по оптимизации	318
1. Используй таблицу подстановки экранных адресов	318
2. Прокручивай только видимое	318
3. Используй PUSH для посимвольного скроллинга	318
4. Разбей посимвольный скроллинг по кадрам	318
5. Трюки с палитрой и атрибутами	319
Итого	319
Глава 18: Игровой цикл и система сущностей	321
18.1 Главный цикл	321
Бюджет кадра: повторение	322
18.2 Конечный автомат игры	323
Определения состояний	323
Таблица переходов	325
Диспетчер	325
Почему не цепочка сравнений?	326
Переходы между состояниями	327
18.3 Ввод: чтение игрока	327
Клавиатура ZX Spectrum	327
Джойстик Kempston	329
Детектирование фронтов: нажатие vs удержание	330
Agon Light 2: PS/2-клавиатура через MOS	330
18.4 Структура сущности	331
Раскладка структуры	331
Почему 10 байт?	332
Почему 16-битный X, но 8-битный Y?	332
Система фиксированной точки 8.8	333
18.5 Массив сущностей	333
Распределение слотов сущностей	334
Перебор сущностей	334
Диспатч обновления по типу	335
Обработчик обновления игрока	335
18.6 Пул объектов	336
Создание пули	337
Деактивация сущности	338
Обработчик обновления пули	338
Размер пула	338
18.7 Сущности взрывов и эффектов	339
18.8 Собираем всё вместе: каркас игры	340
18.9 Agon Light 2: та же архитектура, больше пространства	351
18.10 Проектные решения и компромиссы	352
Фиксированная vs переменная частота кадров	352
Размер сущности: компактный vs щедрый	352
Когда использовать HL вместо IX	353
Итого	353
Попробуй сам	354

Глава 19: Столкновения, физика и ИИ врагов	356
Часть 1: Обнаружение столкновений	356
AABB: единственная форма, которая тебе нужна	356
Порядок тестов для быстрейшего отклонения	360
Столкновения с тайлами: тайлмэп как поверхность столкновений	360
Скользящая реакция на столкновение	363
Часть 2: Физика	364
Гравитация: убедительное падение	364
Почему фиксированная точка важна здесь	365
Прыжок: антигравитационный импульс	365
Прыжки переменной высоты	366
Трение: замедление на земле	367
Применение скорости к позиции	368
Цикл физики	368
Часть 3: ИИ врагов	369
Конечный автомат	369
Таблица JP	369
Patrol: тупой обход	370
Chase: неотступный преследователь	371
Attack: выстрел и перезарядка	373
Retreat: обратное преследование	374
Death: анимация и удаление	375
Оптимизация: обновление ИИ каждый 2-й или 3-й кадр	375
Часть 4: Практика — четыре типа врагов	376
Подключение к игровому циклу	378
Заметки по Agon Light 2	380
Руководство по настройке	380
Итого	381
Попробуй сам	382
Глава 20: Рабочий процесс создания демо — от идеи до компо	384
20.1 Что означает “дизайн” в демо	385
20.2 Lo-Fi Motion: полное исследование случая	385
Концепция: “белорусский пиксель”	385
Таблица сцен	386
Четырнадцать эффектов	386
Сами эффекты	387
Тулчейн	387
Конвейер сборки	388
Временная шкала: две недели вечеров	388
20.3 Культура making-of	389
Eager: технический NFO	389
GABBA: другой рабочий процесс	390
NHBF: головоломка	390
20.4 Тулчейн в деталях	391
Ассемблер: sjasmplus	391
Эмуляторы	391
Графика и генерация кода	392
Автоматизация сборки и CI	392
Синхронизация и композитинг	392
20.5 Культура компо	394

Основные пати	395
Как участвовать в первом компо	395
20.6 Сообщество	396
Hype (hype.retroscene.org)	396
ZXArt (zxart.ee)	397
Pouet (pouet.net)	397
20.7 Управление проектом для создателей демо	397
Минимально жизнеспособное демо	397
Работа с соавторами	398
Отладка и тестирование	398
20.8 Преодоление платформы: "MORE" Introspec'a	399
20.9 Твоё первое демо: практическая дорожная карта	399
Неделя 1: Фундамент	401
Неделя 2: Эффекты	401
Неделя 3: Полировка	402
Неделя 4: Релиз	402
Итого	402
Глава 21: Полная игра - ZX Spectrum 128K	404
21.1 Архитектура проекта	404
Структура каталогов	405
Система сборки	405
21.2 Карта памяти: распределение банков 128K	406
Распределение банков Ironclaw	407
Стек	410
21.3 Конечный автомат	410
21.4 Кадр геймплея	411
Чтение ввода	412
Движок скроллинга	413
21.5 Интеграция спрайтов	414
Восстановление фона (грязные прямоугольники)	415
21.6 Столкновения, физика и ИИ в контексте	416
Цикл физика-столкновения	417
Столкновение с тайлами	417
ИИ врагов	418
21.7 Интеграция звука	420
Музыка	420
Звуковые эффекты	420
21.8 Загрузка: лента и DivMMC	422
Файл .tap и загрузчик на BASIC	422
Загрузка через esxDOS (DivMMC)	423
21.9 Экран загрузки, меню и таблица рекордов	425
Экран загрузки	425
Титульный экран и меню	426
Таблица рекордов	428
21.10 Загрузка уровней и распаковка	429
21.11 Профилирование с DeZog	431
Что такое DeZog?	431
Рабочий процесс профилирования	431
Конфигурация DeZog для Ironclaw	433
21.12 Конвейер данных в деталях	434

Тайлсеты (PNG в пиксельный формат Spectrum)	434
Листы спрайтов (PNG в предварительно сдвинутые данные спрайтов)	434
Карты уровней (Tiled JSON в бинарную тайловую карту)	434
Музыка (Vortex Tracker II в PT3)	435
Сборка воедино	435
21.13 Формат релиза: сборка .tap	435
Тестирование релиза	437
21.14 Финальная полировка	437
Итого	438
Глава 22: Порттирование — Agon Light 2	440
Тот же ISA, другой мир	440
Архитектура с высоты птичьего полёта	441
Режим ADL против Z80-совместимого режима	443
Зачем тебе может понадобиться Z80-режим	443
Механизм переключения режимов	443
Практическое правило	444
Ловушка MBASE	444
Что переносится напрямую	445
Игровая логика и система сущностей	445
Обнаружение столкновений AABB	446
Арифметика с фиксированной точкой	446
Конечный автомат	447
Что требует переписывания	448
Рендеринг: от фреймбуфера к командам VDP	448
Что нужно продумать заново	450
Архитектура памяти	451
Загрузка	451
Звук	452
Ввод	453
eZ80 на 18 МГц: что по-прежнему важно, а что нет	454
Что по-прежнему важно: эффективность внутренних циклов	454
Что становится неактуальным: трюки экономии памяти	455
Что становится неактуальным: самомодифицирующийся код (SMC)	455
Что становится неактуальным: стековые трюки для рендеринга	456
Сравнительная таблица	456
Процесс портирования: пошагово	457
Шаг 1: Настрой проект Agon	457
Шаг 2: Замени слой рендеринга	457
Шаг 3: Транслируй игровую логику	458
Шаг 4: Перепиши звук	458
Шаг 5: Перепиши ввод	458
Шаг 6: Перепиши загрузку	458
Шаг 7: Тестируй и настраивай	458
Чему каждая платформа заставляет тебя учиться	459
Заметка об инструкциях eZ80	459
Итого	460
Глава 23: Z80-разработка с помощью ИИ	461
23.1 Историческая параллель: HiSoft C на ZX Spectrum	461
23.2 Цикл обратной связи Claude Code	462
Цикл	462

Конкретный пример	463
Что делает цикл быстрым	464
Что делает цикл медленным	465
23.3 Интеграция с DeZog: вторая половина цикла	466
Рабочий процесс ИИ + DeZog	466
Инспекция памяти для кода, работающего с данными	466
Что DeZog не может делать (пока)	467
23.4 Когда ИИ помогает, когда нет	467
ИИ помогает: высокая уверенность	467
ИИ помогает: средняя уверенность	467
ИИ не помогает: низкая уверенность	468
23.5 Кейс-стади: создание MinZ	468
Что такое MinZ	469
Где ИИ помог при создании MinZ	469
Где ИИ не помог при создании MinZ	471
Вердикт по MinZ	471
23.5b Врезка: Другой ИИ — супероптимизация полным перебором	472
23.6 Честный взгляд: «Z80 они по-прежнему не знают»	473
Что именно ИИ делает неправильно	473
В чём Introspec прав	475
В чём Introspec не совсем прав	475
23.7 Демо «Antique Toy»: ИИ на практике	475
23.8 Цикл обратной связи на практике	476
23.9 Построение собственного рабочего процесса с ИИ	477
Промпт-инжиниринг для Z80	477
23.10 Общая картина	477
Итого	478
Попробуй сам	478
Глоссарий	480
A. Тайминг и производительность	480
B. Аппаратура — Sinclair и клоны	483
C. Аппаратура — советская/постсоветская экосистема	487
D. Аппаратура — Agon Light 2	488
E. Техники	490
F. Ассемблерная нотация и директивы	495
G. Демосцена и культура	497
Ключевые персоны	499
Ключевые демо и продукции	501
Ключевые публикации	501
H. Алгоритмы и сжатие	501
Приложение А: Краткий справочник инструкций Z80	505
8-битные инструкции загрузки	505
16-битные инструкции загрузки	506
8-битная арифметика и логика	507
16-битная арифметика	509
Сдвиги и вращения	509
Битовые операции	511
Переходы, вызовы, возврат	512
Инструкции ввода/вывода	514
Блочные инструкции	515

Обмен и разное	516
«Быстрые» инструкции демосцены	518
Самый быстрый межрегистровый перенос	518
Самый быстрый способ обнулить регистр	518
Самое быстрое чтение из памяти	518
Самая быстрая запись в память	518
Самая быстрая запись 2 байт	518
Самое быстрое чтение 2 байт	518
Самое быстрое блочное копирование	518
Самый быстрый ввод/вывод	519
Самый быстрый обмен указателей	519
Самый быстрый условный цикл	519
Самый быстрый косвенный переход	519
Недокументированные инструкции	519
IXH, IXL, IYH, IYL (полурегистры индексных регистров)	519
SLL r (логический сдвиг влево)	520
OUT (C),0	520
Недокументированные битовые операции с СВ-префиксом над (IX+d)	520
Шпаргалка по влиянию на флаги	521
Инструкции, устанавливающие все арифметические флаги (S, Z, H, P/V, N, C)	521
Инструкции, устанавливающие Z и S (но НЕ перенос)	521
Инструкции, устанавливающие ТОЛЬКО флаги, связанные с переносом	521
Инструкции, НЕ устанавливающие флаги	521
Практические трюки	521
Архитектура регистров	522
Основной набор регистров	522
Специальные регистры	522
Теневые регистры	522
Привязка регистровых пар к инструкциям	523
Типичные последовательности инструкций	523
Вычисление адреса пикселя (экранный адрес из Y,X)	523
DOWN_HL: сдвиг на одну строку пикселей вниз	524
Беззнаковое умножение 8x8 (сдвигом и сложением)	524
Запись в регистр AY	525
16-битное сравнение (HL с DE)	525
Заливка экрана через стек	525
Быстрый перебор строк пикселей (раздельные счётчики)	526
Таблица быстрых сравнений стоимости	526
Справочник по размеру кодировки инструкций	527
Приложение В: Генерация таблиц синусов и тригонометрические таблицы	529
Стандартный формат	529
Сравнение подходов	530
Подход 1: Полная 256-байтная таблица	531
Подход 2: Четвертьволновая таблица	531
Подход 3: Параболическая аппроксимация (метод Dark'a)	532
Подход 4: Кодирование вторыми разностями (глубокий трюк)	533
Ключевое наблюдение	533

Полная таблица через 2-битные вторые разности	533
Четвертьволна через 2-битные вторые разности	533
Подход 5: Аппроксимация Бхаскары I (VII век)	535
Формула	535
Точность	536
Реализация на Z80	536
Бхаскара I + битовая карта коррекции (точно)	536
Когда использовать Бхаскару I	537
Практические рекомендации	537
Дерево решений	538
Что не работает	538
Заповеди Raider'a	538
Справочник: Полная 256-байтная таблица	539
Приложение С: Краткий справочник по сжатию	541
Сравнительная таблица упаковщиков	541
Дерево решений: какой упаковщик?	546
Степень сжатия типичных данных ZX Spectrum	547
Трюки перед сжатием	550
Минимальный RLE-распаковщик	551
Стандартный распаковщик ZX0 (Z80)	552
Паттерны интеграции	554
Паттерн 1: Распаковка на экран при старте	554
Паттерн 2: Распаковка в буфер между эффектами	554
Паттерн 3: Потоковая распаковка во время воспроизведения	554
Паттерн 4: Сжатые данные с переключением банков (128K)	554
Конвейер сборки: от ресурса к бинарнику	555
Быстрые формулы	556
См. также	556
Приложение D: Настройка среды разработки	557
1. Ассемблер: sjasmplus	557
Установка из исходников	557
Привязка версии	558
Ключевые флаги	558
Расширение файла	558
Шестнадцатеричная запись	559
2. Редактор: VS Code	559
Необходимые расширения	559
Задача сборки	559
Рекомендуемые настройки	560
3. Эмуляторы	561
ZEsarUX — полнофункциональная отладка	561
Fuse — лёгкий и точный	561
Unreal Speccy — Windows, ориентация на Pentagon	562
Для Agon Light 2	562
Какой эмулятор выбрать?	562
4. Отладчик: DeZog	562
Установка	563
Подключение к ZEsarUX	563
Использование встроенного симулятора	563
Ключевые возможности DeZog	564

5. Сборка примеров из книги	564
Клонирование репозитория	564
Предварительные требования	565
Команды сборки	565
Запуск примера	565
6. Структура проекта для собственного кода	566
Минимальный Makefile	566
Соглашение об включениях	567
7. Альтернативные инструменты	567
Другие ассемблеры	567
Другие расширения VS Code	568
CSpect — эмулятор для Next	568
SpectrumAnalyzer	568
8. Устранение неполадок	569
«sjasmplus: command not found»	569
Ошибки компиляции в примерах из книги	569
DeZog не может подключиться к ZEsarUX	569
Эмулятор показывает мусор на экране	569
Выходной файл сборки пуст или нулевого размера	569
Справочник инструментов	569
См. также	570
Приложение Е: Краткий справочник по eZ80	571
1. Обзор архитектуры	571
2. Система режимов	572
Два режима	572
Суффиксы режима	572
Вызовы и переходы с переключением режима	573
Практическое правило	573
Ловушка MBASE	573
3. Новые инструкции	573
Арифметика и тесты	573
Вычисление адресов	574
Ввод-вывод и системные инструкции	574
4. MLT — Революционная инструкция	575
Что даёт MLT	576
Ограничения MLT	576
5. Сводка ключевых различий	576
6. Особенности Agon Light 2	577
Аппаратная часть	577
Бюджет кадра	577
API MOS	578
Протокол команд VDP	578
7. Чек-лист портирования	579
См. также	580
Приложение F: Варианты Z80 — Расширенные наборы инструкций	581
1. Генеалогическое древо Z80	581
2. Z80N — Список желаний демосценера	582
Навигация по экрану (проблема DOWN_HL)	582
Рендеринг спрайтов (проблема маскированного блита)	583
Арифметика (проблема умножения)	584

Битовые манипуляции	584
Циклические сдвиги (проблема многобитного сдвига)	585
Удобные инструкции	586
Общая картина	586
3. eZ80 — Корпоративное расширение	587
4. R800 — Скоростной монстр MSX turboR	588
5. Советские клоны — За железным занавесом	589
6. Сравнительная таблица	589
Эффективная производительность умножения	590
7. Что это значит для книги	590
См. также	591
Приложение G: Справочник регистров AY-3-8910 / TurboSound / Triple AY 592	
Порты ввода/вывода на ZX Spectrum 128K	592
Последовательность записи	592
Чтение регистра	593
Массовая запись регистров	593
Полная карта регистров	593
Обзор	593
R0-R1: Период тона канала А (12 бит)	595
R2-R3: Период тона канала В (12 бит)	595
R4-R5: Период тона канала С (12 бит)	595
R6: Период шума (5 бит)	595
R7: Управление микшером (самый важный регистр)	596
R8-R10: Регистры громкости (5 бит)	597
R11-R12: Период огибающей (16 бит)	598
R13: Форма огибающей (4 бита, только запись)	598
Преобразование периода тона в частоту	601
Формула	601
Тактовая частота AY по платформам	601
Полная таблица частот нот	601
Компактная таблица нот для Z80-кода	604
Таблица #5: Натуральный строй (чистая интонация)	605
TurboSound: 2 x AY	607
Выбор чипа	607
Архитектура движка TurboSound	608
Стерео-конфигурация	608
ZX Spectrum Next: Triple AY (3 x AY)	609
Выбор чипа на Next	609
Стерео-панорамирование каждого канала	609
Типичные паттерны в Z80-коде	610
Заглушить AY	610
Воспроизвести одну ноту на канале А	610
Запустить ноту buzz-bass	611
Перезапуск огибающей	611
Удар цифрового барабана (4-битный ЦАП)	611
Замечания по форматам трекеров	612
ProTracker 3 (.pt3)	612
Другие форматы трекеров	613
Vortex Tracker II	614

AY-3-8910 vs YM2149: различия, которые имеют значение	614
Краткая справочная карточка	615
Сводка регистров (отрывная)	615
Приложение Н: API хранилищ — TR-DOS и esxDOS	617
1. TR-DOS (Beta Disk 128)	617
Аппаратная часть	617
Формат диска	617
Карта портов WD1793	618
Команды WD1793	618
API ПЗУ: загрузка файла	619
Прямой доступ к секторам	620
Обнаружение диска	621
2. esxDOS (DivMMC / DivIDE)	621
Аппаратная часть	621
Паттерн API	621
Справочник функций	622
Режимы открытия файла	623
Значения Seek Whence	623
Пример кода: загрузка файла	623
Пример кода: потоковое чтение данных из файла	624
Обнаружение esxDOS	625
3. +3DOS (Amstrad +3)	626
4. Практические паттерны	626
Загрузка экрана с диска (TR-DOS)	626
Загрузка экрана с SD (esxDOS)	626
Двухрежимный загрузчик	627
Потоковое чтение сжатых данных	627
5. Справочник форматов файлов	628
6. См. также	629
Приложение I: Bytebeat и AY-Beat — генеративный звук на Z80	630
1. Классический Bytebeat: традиция PCM	630
Знаменитые формулы	631
Почему это работает	631
На Spectrum: тупик бипера	631
2. AY-Beat: bytebeat, переосмысленный для тонового генератора	632
Базовая архитектура AY-Beat	632
Что меняется по сравнению с PCM-bytebeat	633
3. Дрон: огибающая + тон (режим E+T)	634
Рецепт дрона	634
Стоймость в байтах	636
4. Шумовая перкуссия	636
Базовый паттерн бочки	636
Характер перкуссии в зависимости от периода шума	637
Ритмическое разнообразие через битовые маски	637
Использование огибающей для затухания барабана	637
5. Многоканальная гармония	638
Три голоса из одной формулы	638
Почему ротация создаёт гармонию	639
Формулы громкости для многоканальности	639
6. Кулинарная книга формул	640

Формулы периода тона	640
Формулы громкости	642
Как читать таблицу	642
7. Собираем вместе: полный движок AY-Beat	643
Что это производит	645
Точки настройки	645
8. Продвинутое: комбинирование техник	645
Архитектура для 256-байтного интро	646
Бюджет регистров	646
Расклад по размеру	646
9. AY-как-ЦАП: классический bytebeat через регистр громкости	647
Три пути вывода в сравнении	648
10. Теория музыки для алгоритмов	648
Таблицы гамм: ограничение выхода приятными нотами	648
Октачная деривация: бесплатный диапазон высоты	649
Арпеджио: тоны аккорда последовательно	650
Пошаговые орнаменты: трели, морденты и глиссандо	650
Аkkордовые последовательности: гармоническое движение	651
Суммарный бюджет данных для богатой музыки	652
11. L-системные грамматики: фрактальные мелодии	652
Концепция	652
Почему L-системы работают для музыки	653
Полезные правила L-систем	653
Реализация на Z80	653
Мелодия как движение, а не абсолютные ноты	655
Трибоначки: три символа для более богатых паттернов	656
Мелодии PRNG с отобранными сидами	656
Комбинирование L-систем с другими техниками	657
Другие грамматики для музыки	658
См. также	658
Приложение J: Современные инструменты для создания ретро-демо	659
J.1 Два мира, одна философия	659
J.2 Инструменты синхронизации	660
Проблема синхронизации	660
GNU Rocket	660
Vortex Tracker II	661
Blender VSE (Video Sequence Editor)	662
Blender Graph Editor	663
Motion Canvas	664
Сравнительная таблица	664
J.3 Предвизуализация и раскадровка	664
Blender как инструмент раскадровки	664
Видеоредакторы для черновой сборки	665
Рабочий процесс GABBA	665
Kolnogorov о планировании синхронизации	665
J.4 Генераторы данных	666
Unity как генератор данных	666
Unreal Engine как генератор данных	666
Blender как генератор данных	667
Конвейер экспорта	667

Когда использовать что	668
J.5 Инструментарий РС-демосцены: краткая история	668
Farbrausch (1999-2012)	668
TiXL (2024-наши дни)	669
Bonzomatic	670
Crinkler	670
Squishy	671
Shader Minifier	671
z80-optimizer	671
Общая философия	671
J.6 Музыкальные и звуковые инструменты	672
Sointu	672
4klang	672
WaveSabre	672
Oidos	673
Furnace	673
Сравнительная таблица	673
J.7 Практические рецепты	674
Рецепт 1: GNU Rocket → таблица синхронизации Z80	674
Рецепт 2: Blender VSE → таблица номеров кадров	675
Рецепт 3: Unity VR → данные траектории	675
Рецепт 4: Furnace → музыка для AY	677
Рецепт 5: packbench → предварительный анализ перед сжатием .	678
Дополнительное чтение	679

!!! 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.

Глава 1: Мыслить тактами

«Кодерские эффекты — это всегда про эволюцию вычислительной схемы.» – Introspec (spke), Life on Mars

У тебя есть 71 680 тактов. Это твой холст, твой бюджет, весь твой мир. Каждая инструкция, которую ты пишешь, стоит часть этих тактов. Каждый кадр счётчик обнуляется, и ты получаешь ещё 71 680 — ни больше, ни меньше. Не уложился в дедлайн — экран рвётся, музыка заикается, иллюзия рушится.

Эта глава о том, как научиться видеть свой код глазами демосценера на Z80: не как текст, не как алгоритм, а как *бюджет*.

T-state: Валюта Z80

Процессор Z80 не выполняет инструкции одинаковыми порциями. Каждая инструкция занимает определённое число **тактов** (T-state) — элементарных тактовых циклов процессора. На частоте 3,5 МГц один такт длится примерно 286 наносекунд. Это число не важно. Важно то, что инструкции стоят очень по-разному, и эти стоимости нужно знать наизусть.

Вот горстка инструкций, которые ты будешь использовать постоянно:

Инструкция	Что делает	Такты
NOP	Ничего	4
LD A,B	Копирует B в A	4
LD A,(HL)	Загружает байт по адресу в HL	7
LD (HL),A	Записывает A по адресу в HL	7
LD A,n	Загружает непосредственный байт в A	7
INC HL	Инкремент HL	6
ADD A,B	Складывает B с A	4
PUSH HL	Кладёт HL на стек	11
DJNZ label	Декремент B, переход если не ноль	13 (переход) / 8 (проваливается)
LDIR	Блочное копирование, за байт	21 (повтор) / 16 (последний байт)

Инструкция	Что делает	Такты
OUT (n), A	Пишет А в порт ввода-вывода	11

Посмотри на разброс. Регистровое LD A,B стоит 4 такта — минимум для любой инструкции. Чтение из памяти LD A, (HL) стоит 7, потому что процессору нужны дополнительные машинные циклы, чтобы выставить адрес на шину и дождаться ответа от ОЗУ. LDIR, инструкция блочного копирования, к которой каждый спектрумист тянется инстинктивно, стоит 21 такт за копируемый байт (кроме последнего, который стоит 16). Это более чем в пять раз дороже NOP.

Z80 Instruction Costs

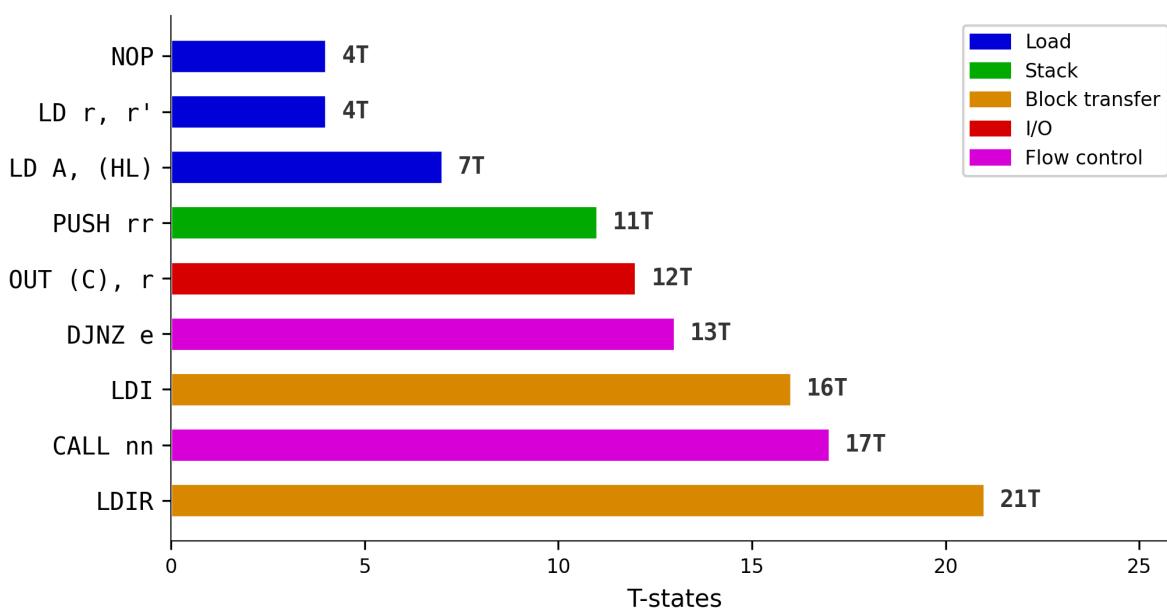


Рис. 1: T-state costs for common Z80 instructions

Почему это важно? Потому что когда ты заполняешь экран, обновляешь данные спрайтов или вычисляешь следующий кадр плазмы, каждая инструкция отъедает от бюджета. Разница между инструкцией в 4 такта и в 7 тактов, умноженная на десять тысяч итераций внутреннего цикла — это разница между эффектом, работающим на 50 кадрах в секунду, и эффектом, который не успевает.

Машинные циклы и доступ к памяти

Каждый такт (T-state) — это один тик процессорного генератора, но Z80 не обращается к памяти на каждом такте. Инструкции разбиты на **машинные циклы** (M-циклы), каждый из которых занимает 3-6 тактов. Первый машинный цикл каждой инструкции — это **выборка опкода** (M1), которая всегда занимает 4 такта: процессор выставляет счётчик команд на адресную шину, считывает байт опкода и одновременно обновляет DRAM (регенерация). Последующие машинные циклы считывают дополнительные байты (операнды, данные из памяти) или записывают результаты.

Вот почему LD A,B занимает ровно 4 такта — это однобайтовая инструкция, которая целиком завершается в фазе выборки опкода. А LD A,(HL) занимает 7 тактов: 4 на выборку опкода, затем ещё 3 на цикл чтения памяти, когда процессор выставляет HL на адресную шину и считывает байт по этому адресу.

Не нужно запоминать внутреннюю разбивку машинных циклов каждой инструкции. Но понимание паттерна — выборка опкода + чтение операндов + обращения к памяти = общая стоимость — помогает выработать интуицию, почему инструкции стоят столько, сколько стоят. PUSH HL в 11 тактов обретает смысл, когда понимаешь, что процессору нужно выполнить выборку опкода (5Т в данном случае, потому что одновременно декрементируется SP), а затем два отдельных цикла записи в память (по 3Т каждый) для сохранения старшего и младшего байтов HL на стеке.

Кадр: Твой холст

ZX Spectrum формирует PAL-видеосигнал с частотой примерно 50 кадров в секунду. Каждый кадр микросхема ULA считывает видеопамять и отрисовывает экран строка за строкой. В конце каждого кадра ULA генерирует маскируемое прерывание. Процессор выполняет инструкцию HALT, чтобы дождаться этого прерывания, делает свою работу, а затем снова HALT — ждёт следующий кадр. Это сердцебиение каждой программы для Spectrum.

Число тактов между одним прерыванием и следующим — **бюджет кадра** — зависит от машины:

Машина	Тактов на кадр	Строк развертки	Гц
ZX Spectrum 48K	69 888	312	50,08
ZX Spectrum 128K	70 908	311	50,02
Pentagon 128	71 680	320	48,83

Это *общее* число тактов между прерываниями. Практический бюджет меньше — вычти стоимость обработчика прерывания (музыкальный плеер РТЗ обычно потребляет 3 000–5 000 тактов на кадр), накладные расходы на HALT и, на машинах не-Pentagon, штрафы за спорную память. На Pentagon с музыкальным плеером рассчитывай примерно на 66 000–68 000 тактов для основного цикла. В главе 15 есть подробные тактовые карты.

Если твой основной цикл — обработка ввода, игровая логика, обновление звука, отрисовка экрана — занимает больше тактов, чем один кадр, ты теряешь кадры. Всё замедляется. Трюк с полоской бордюра, который мы построим далее в этой главе, сделает это болезненно наглядным.

Для масштаба: одна LDIR, копирующая 6 912 байт (полный экран пиксельных данных), стоит примерно $6\ 912 \times 21 = 145\ 152$ такта. Это больше двух целых кадров на 48K Spectrum. Ты не можешь даже скопировать экран целиком за один кадр простейшим способом. Именно такие ограничения заставляют быть изобретательным.

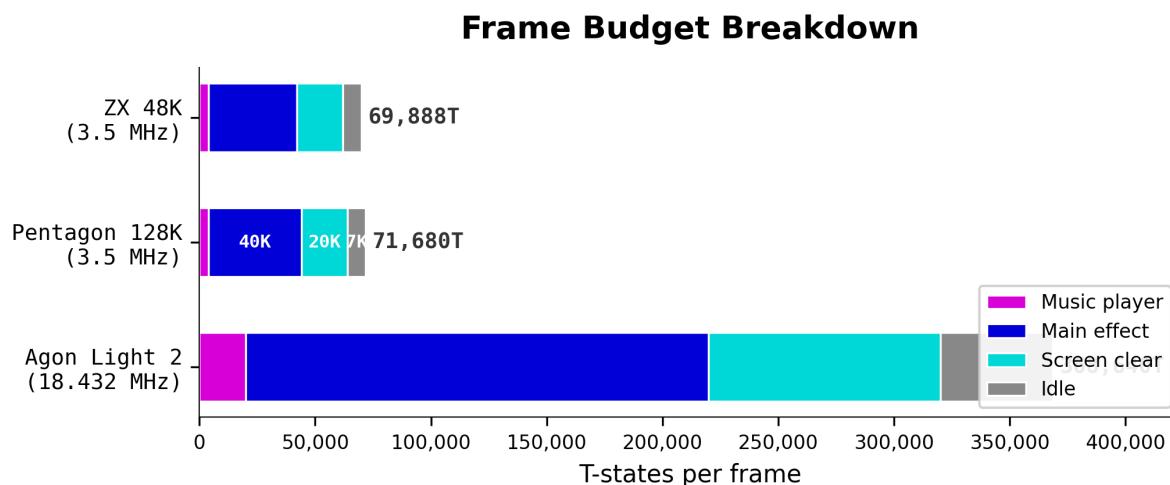


Рис. 2: Frame budget breakdown across ZX Spectrum models

Pentagon против «ждущих» машин

Ты заметил, что бюджеты кадра в таблице выше различаются. Разница не только в числах — она отражает фундаментальное архитектурное расхождение, которое определило ZX Spectrum демосцену.

Оригинальные машины Sinclair

На оригинальных 48K и 128K Spectrum экранная память расположена по адресам \$4000–\$5AFF (пиксельные данные) и \$5800–\$5B00 (атрибуты цвета). Этот регион памяти — а фактически весь диапазон \$4000–\$7FFF — является **спорной памятью** (contended memory). ULA (Uncommitted Logic Array), формирующая видеосигнал, должна читать эту память для отрисовки экрана. Процессор и ULA разделяют одну шину памяти, и когда оба хотят читать одновременно, ULA побеждает. Процессор вынужден ждать.

Во время отрисовки 192 активных строк экрана каждое обращение процессора к диапазону \$4000–\$7FFF потенциально задерживается. Задержка следует повторяющемуся 8-тактовому паттерну: 6, 5, 4, 3, 2, 1, 0, 0 дополнительных тактов ожидания, циклически по каждой строке развёртки. Инструкция, которая должна занимать 7 тактов, может занять 13, если попадёт на худшую фазу цикла конкуренции.

Это превращает подсчёт тактов на оригинальных Spectrum в кошмар. Тщательно просчитанный внутренний цикл работает с разной скоростью в зависимости от того, где в кадре он выполняется и попадает ли его код или данные в спорный диапазон. Introspec задокументировал это в статьях «GO WEST» на Нуре (2015): во время отрисовки экрана каждое обращение к спорной памяти обходится в среднем в 2,625 дополнительных такта. Для стековых операций, пишущих в экранную память, рассчитывай примерно на 1,3 дополнительных такта на байт.

Pentagon: Чистый тайминг

Pentagon 128, самый популярный советский клон ZX Spectrum, пошёл другим путём. Его разработчики дали ULA собственное окно доступа к памяти, не конфликтующее с процессором. **На Pentagon нет спорной памяти.** Каждая инструкция занимает ровно столько тактов, сколько указано в даташите, независимо от того, где находится код и к какой памяти он обращается.

Вот почему у Pentagon другая длина кадра — 71 680 тактов, 320 строк развертки. Тайминг ULA немного отличается, потому что нет необходимости чередовать доступ CPU и ULA. Но выигрыш огромен: ты можешь считать такты с абсолютной уверенностью. Когда твой внутренний цикл заявляет 36 тактов на итерацию — он стоит 36 тактов на итерацию, каждый раз, в любом месте кадра.

Этот чистый тайминг — причина того, что Pentagon стал стандартной платформой для ZX Spectrum демосцены, особенно на постсоветском пространстве, где эти клоны были повсеместны. Когда ты смотришь демо от групп X-Trade, 4D+TBK (Triebkraft) или Life on Mars — они в подавляющем большинстве рассчитаны на тайминг Pentagon. Когда Introspec писал свой легендарный технический разбор Illusion от X-Trade, он указывал тактовые подсчёты для Pentagon.

Для обучения модель Pentagon идеальна: можно сосредоточиться на понимании стоимости инструкций, не беспокоясь об эффектах конкуренции. Все таблицы тактов в этой книге предполагают тайминг Pentagon, если не указано иное. Когда потребуется обсудить различия (а мы это сделаем в главе 15), мы скажем об этом явно.

Практическое правило: размещай критичный по времени код в неспорной памяти (\$8000–\$FFFF на 48K), и твои подсчёты тактов будут верны как на Pentagon, так и на оригинальных Spectrum.

Мыслить бюджетами

Теперь, когда ты знаешь размер кадра, можно начинать арифметику, которая определяет мышление Z80-демосценера.

Допустим, ты хочешь каждый кадр заполнять весь экран вычисленным цветом — простой эффект плазмы, обновляя только 768 байт области атрибутов по адресу \$5800. На 50 fps тебе нужно вычислить и записать 768 цветовых значений за каждые 71 680 тактов.

Если внутренний цикл на один байт атрибута выглядит так:

```

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

```

Это 32 такта на байт. На 768 байт: $32 \times 768 = 24\,576$ тактов. Добавь накладные расходы цикла (поддержание счётчиков строк и столбцов, DJNZ для внутреннего цикла), и получится примерно 28 000–30 000 тактов. Остаётся более 40 000 тактов на всё остальное — проигрывание музыки, обработку ввода, что угодно.

Но что если ты захочешь обновить каждый пиксельный байт — все 6 144? При 32 тактах на байт это 196 608 тактов — почти три кадра. Внезапно ты смотришь на частоту обновления 17 fps вместо 50 fps. Тебе нужен либо более быстрый внутренний цикл, либо меньшая область обновления, либо совершенно другой подход.

Так думают программисты Z80. Каждое проектное решение начинается с арифметики: сколько байт, сколько тактов на байт, сколько тактов в бюджете кадра, уложится ли? Когда не укладывается, ты тянешься не к более быстрой машине — а к более хитрому алгоритму.

Врезка: Agon Light 2

Agon Light 2 работает на Zilog eZ80 с частотой 18,432 МГц. eZ80 выполняет тот же набор инструкций Z80 (это прямой архитектурный потомок), но большинство инструкций выполняются за меньшее число тактов — многие однобайтовые инструкции завершаются за 1 такт вместо 4. На частоте 18,432 МГц при 50 Гц ты получаешь приблизительно **368 640 тактов на кадр**.

Это чуть больше, чем в 5 раз превышает бюджет Pentagon. Тот же язык ассемблера Z80, те же регистры, те же мнемоники инструкций — но в пять раз больше пространства для манёвра. Внутренний цикл, потребляющий 70% кадра Pentagon, может занять лишь 14% кадра Agon.

Это не делает Agon «лёгким». У него свои ограничения: нет видеопамяти в стиле ULA (дисплеем управляет сопроцессор ESP32, выполняющий VDP), плоская 24-битная адресация вместо банковой памяти и совершенно другая модель ввода-вывода. Но если ты когда-нибудь мечтал о чуть *большем запасе* в бюджете кадра для чего-то амбициозного, Agon — это место, где то же мышление Z80 масштабируется вверх.

На протяжении всей книги мы будем отмечать, где больший бюджет Agon меняет расклад. Пока просто запомни число: **~368 000 тактов**. Тот же ISA, в пять раз больше холст.

Практика: Настройка среды разработки

Прежде чем писать первую тестовую обвязку, тебе нужен рабочий набор инструментов. Описанная здесь настройка следует руководству sq с Нуле (2019), ставшему стандартом сообщества.

Что понадобится

1. **VS Code** — редактор и интегрированная среда.
2. **Z80 Macro Assembler extension** от mborik (`mborik.z80-macroasm`) — подсветка синтаксиса, автодополнение, разрешение символов для ассемблера Z80. Устанавливается из магазина VS Code.
3. **Z80 Assembly Meter** от Nestor Sancho — показывает количество байт и тактов выделенной инструкции (или блока инструкций) в строке состояния. Незаменим. Выдели блок кода и мгновенно увидишь его суммарную стоимость в тактах.
4. **sjasmplus** — собственно ассемблер. Кроссплатформенный, с открытым исходным кодом, поддерживает макросы, скрипты Lua, множество выходных форматов. Скачай с <https://github.com/z00m128/sjasmplus> и размести бинарник в PATH.
5. **Unreal Speccy** (Windows) или **Fuse** (кроссплатформенный) — эмулятор. Unreal Speccy предпочтителен для разработки демо, потому что точно эмулирует тайминг Pentagon и имеет встроенный отладчик.

Структура проекта

Создай каталог для экспериментов по главе 1. Минимальный проект выглядит так:

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

Конфигурация сборки

В VS Code настрой задачу сборки (`.vscode/tasks.json`), чтобы компилировать по Ctrl+Shift+B:

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

```

    "file": 1,
    "line": 2,
    "severity": 3,
    "message": 4
  }
}
]
}
```

Нажми Ctrl+Shift+B. Если sjasmplus в PATH и нет ошибок, ты получишь файл .sna или .tap (в зависимости от директив в исходнике), который можно открыть прямо в эмуляторе.

Для интеграции с Unreal Speccy расширение Alex_Rider (2024) добавляет привязку F5-для-запуска — эмулятор автоматически открывает скомпилированный снапшот. Если ты на macOS или Linux и используешь Fuse, то же самое делает простое правило в Makefile:

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

Практика: Тестовая обвязка

Это самый важный инструмент отладки, который ты построишь за всю книгу. Он до абсурда прост, не требует специального оборудования, и ты будешь использовать его постоянно.

Идея: сменить цвет бордюра на красный непосредственно перед измеряемым кодом и обратно на чёрный сразу после. Бордюр Spectrum отрисовывается ULA в реальном времени, синхронно с электронным лучом. Более широкая красная полоса означает больше тактов, потраченных в твоём коде.

Вот полная обвязка:

```

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

```

Загрузи это в эмулятор. Ты увидишь красную полосу поперёк бордюра. Высота этой полосы прямо пропорциональна числу тактов, затраченных тестируемым кодом.



Рис. 3: Вывод тестовой обвязки — красные полосы бордюра показывают такты (T-state), затраченные тестируемым кодом, чёрные промежутки показывают время простоя

Чтение полосы

Каждая строка развёртки занимает 224 такта (на Pentagon). Если твоя красная полоса высотой N строк, твой код занял примерно $N \times 224$ тактов. Пример выше использует около 7 419 тактов, что составляет примерно 33 строки — ты увидишь красную полосу примерно на одну шестую высоты бордюра.

Теперь попробуй заменить цикл NOP чем-то потяжелее. Замени четыре NOP на:

```
.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

```

Красная полоса заметно вырастет. Эта визуальная разница — без отладчика, без профайлера, вообще без инструментов — составляет 2 816 тактов. Около 12 строк развёртки.

Именно так кодеры демо на Spectrum профилировали свои эффекты с 1980-х. Бордюр — твой осциллограф.

Вариации

Можно использовать разные цвета для маркировки разных фаз кода:

```

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

```

Теперь бордюр показывает красную полосу (отрисовка спрайтов), затем синюю (музыка), затем зелёную (игровая логика), затем чёрную (времяостоя). Одним взглядом видно, какая подсистема пожирает бюджет кадра.

Замечание о `xor a` против `ld a, 0`: оба устанавливают A в ноль. XOR A занимает 4 такта и 1 байт. LD A, 0 — 7 тактов и 2 байта. В тестовой обвязке разница ничтожна, но это стоит заметить — именно из такой микро-осведомлённости и состоит программирование на Z80.

Что помещается в кадр?

Давай используем бюджетную арифметику, чтобы ответить на несколько практических вопросов.

Сколько спрайтов можно нарисовать за кадр? Маскированный спрайт 16x16 методом OR+AND занимает примерно 16 строк x (прочитать маску + прочитать спрайт + прочитать экран + комбинировать + записать на экран) на байт. Разумная оценка — около 1 200 тактов на спрайт. На Pentagon это $71\,680 / 1\,200 = \sim 59$ спрайтов, если отрисовка — *единственное*, чем ты занимаешься. На практике, с музыкой, игровой логикой и всем прочим, 8-12 полноразмерных спрайтов за кадр — типичная цифра.

Сколько байт может скопировать LDIR за кадр? При 21 такте на байт: $71\,680 / 21 = 3\,413$ байт. Даже не половина экрана.

Сколько умножений? Быстрое табличное умножение 8x8 занимает около 54 тактов. $71\ 680 / 54 = 1\ 327$ умножений за кадр. Для вращения одной 3D-точки нужно 9 умножений. Значит, можно повернуть примерно 147 точек за кадр, если больше ничего не делать. Практический предел при полноценном демо-движке: 30–50 точек.

Каждый проектный вопрос сводится к этой арифметике. Смогу ли я это сделать? Сколько штук? От чего придётся отказаться, чтобы освободить место?

Историческая заметка: Совет Dark'a

В 1997 году программист по имени Dark из группы X-Trade опубликовал серию статей в *Spectrum Expert #01*, российском электронном журнале для разработчиков ZX Spectrum. Эти статьи охватывали умножение, деление, генерацию синуса/косинуса и алгоритмы рисования линий на ассемблере Z80 — фундаментальные строительные блоки, на которых работает каждый демо-эффект.

Dark начал с такого совета:

«Прочитайте учебник математики — производные, интегралы. Зная их, вы сможете создать таблицу практически любой функции в ассемблере.»

Это была не пустая теория. Dark был не просто автором — он был кодером. Демо *Illusion* от X-Trade, выпущенное на ENLiGHT'96, включало текстурированную вращающуюся сферу, ротозумер, 3D-движок и прыгающий точечный скроллер — и всё это работало на Z80 с частотой 3,5 МГц. Алгоритмы, которые Dark описывал в журнальных статьях, были теми же алгоритмами, которые приводили в движение эффекты его демо.

Двадцать лет спустя Introspec (spke) опубликовал детальный технический разбор *Illusion* на Hype, анализируя внутренние циклы инструкция за инструкцией, подсчитывая каждый такт. Журнальные статьи 1997 года и обратная разработка 2017 года рассказывают одну и ту же историю с двух сторон: автор объясняет свои строительные блоки, а коллега измеряет готовую машину. Мы будем следовать этой нити на протяжении всей книги.

Совет Dark'a остаётся в силе: математика не опциональна. Не нужна степень в математике, но нужно понимать, как превратить математическую функцию в таблицу, как приближать дорогие операции дешёвыми и как думать о компромиссе между точностью и скоростью. Глава 4 детально разберёт алгоритмы Dark'a. Пока запомни его совет. Это отправная точка всего.

Вычислительная схема

Introspec, размышляя о том, что делает хороший демо-эффект, свёл философию в одно предложение:

«Кодерские эффекты — это всегда про эволюцию вычислительной схемы.»

Это самый глубокий тезис этой главы. Демо-эффект — это не картинка, это *процесс*. Каждый кадр вычислительная схема порождает следующее состояние из предыдущего. Искусство — в выборе схемы, которая производит визуально убедительную эволюцию и при этом укладывается в бюджет кадра.

Плазма — это вычислительная схема: суммирование синусоид в каждой точке сетки со сдвигом по времени. Туннель — это вычислительная схема: выборка угла и расстояния из предвычисленных таблиц со сдвигом по времени. Вращающийся 3D-объект — это вычислительная схема: умножение координат вершин на матрицу вращения, которая меняется каждый кадр. Конкретная схема определяет визуальный результат, стоимость в тактах и требования к памяти — всё сразу, всё взаимосвязано.

Когда ты садишься писать эффект, ты спрашиваешь не «как мне нарисовать эту картинку». Ты спрашиваешь «какое вычисление, эволюционирующее кадр за кадром, порождает этот визуал?» Этот сдвиг мышления — от изображения к процессу, от результата к схеме — и есть мировоззрение Z80-программиста.

И первое ограничение для любой схемы — бюджет. 71 680 тактов. Сможешь ли ты развернуть своё вычисление в рамках этого бюджета? Если нет — можешь ли найти более дешёвую схему, дающую похожий визуал? Можешь ли предвычислить часть схемы в таблицы? Можешь ли распределить вычисление на несколько кадров? Можешь ли использовать симметрию, чтобы вычислить половину экрана и отзеркалить вторую?

Эти вопросы движут каждой главой этой книги. Они начинаются здесь, с подсчёта тактов.

Итого

- У каждой инструкции Z80 есть определённая стоимость в тактах. Выучи основные наизусть: NOP = 4, LD A,B = 4, LD A,(HL) = 7, PUSH HL = 11, LDIR = 21/16, OUT (n),A = 11.
 - **Бюджет кадра** — твоё жёсткое ограничение: 69 888 тактов (48K), 70 908 (128K) или 71 680 (Pentagon). На 50 fps всё должно уложиться.
 - **Pentagon не имеет спорной памяти**, что делает подсчёт тактов надёжным и предсказуемым. Именно поэтому он стал стандартом демосцены.
 - **Agon Light 2** (eZ80 @ 18,432 МГц) даёт ~368 000 тактов на кадр — тот же набор инструкций, в пять раз больше пространства.
 - **Тестовая обвязка с цветом бордюра** — твой осциллограф: красный перед, чёрный после, читай ширину полосы.
 - Программирование Z80 — это **бюджетная арифметика**: байты x тактов на байт против бюджета кадра. Каждое проектное решение начинается отсюда.
 - Эффекты — это **вычислительные схемы, эволюционирующие во времени**. Искусство — в нахождении схемы, которая укладывается в бюджет и хорошо выглядит.
-

Попробуй сам

1. Собери тестовую обвязку из этой главы. Замени цикл NOP на LDIR, копирующую 256 байт, и сравни ширину полосы с NOP-версией. Рассчитай ожидаемую разницу в тактах и проверь её визуально.
2. Напиши цикл, заполняющий все 768 байт области атрибутов (\$5800–\$5AFF) одним значением цвета. Измерь его обвязкой. Теперь попробуй заполнить с помощью LDIR вместо побайтового цикла. Что быстрее? На сколько строк развёртки?
3. Открой Z80 Assembly Meter в VS Code. Выделяй разные блоки кода и следи за счётчиком тактов в строке состояния. Привыкни проверять стоимость на ходу.
4. Настрой многоцветный бордюрный профайлер (красный / синий / зелёный / чёрный) с тремя фиктивными циклами разной длины. Подбери количество итераций, пока не сможешь визуально различить все три полосы. Это калибровочное упражнение для чтения тайминга по бордюру.

Далее: Глава 2 — Экран как головоломка. Мы погрузимся в печально известную перепутанную раскладку видеопамяти Spectrum и узнаем, почему INC H перемещает на один пиксель вниз.

Глава 2: Экран как головоломка

«Почему строки идут в таком порядке?» – Каждый программист ZX Spectrum, рано или поздно

Открой любой эмулятор, набери PEEK 16384 — и ты читаешь первый байт экранной памяти Spectrum. Но какой это байт? Не верхний левый экрана в каком-либо простом смысле. Пиксель с координатой (0,0) действительно там — но пиксель (0,1), следующая строка вниз, живёт на 256 байт дальше. Пиксель (0,8), верхняя строка второго знакоместа, расположен всего в 32 байтах от начала. А пиксель (0,64) — первая строка средней трети экрана — живёт ровно в 2 048 байтах от начала, по адресу \$4800.

Это самая знаменитая головоломка Spectrum. Раскладка экранной памяти не последовательная, не интуитивная и не случайная. Она — следствие аппаратных решений, принятых в 1982 году, и она определяет каждый фрагмент кода, который работает с экраном. Понимание этой раскладки — и изучение трюков, позволяющих быстро по ней перемещаться — фундаментально для всего, что последует в этой книге.

Карта памяти: 6 912 байт экрана

Экран Spectrum занимает фиксированную область памяти:

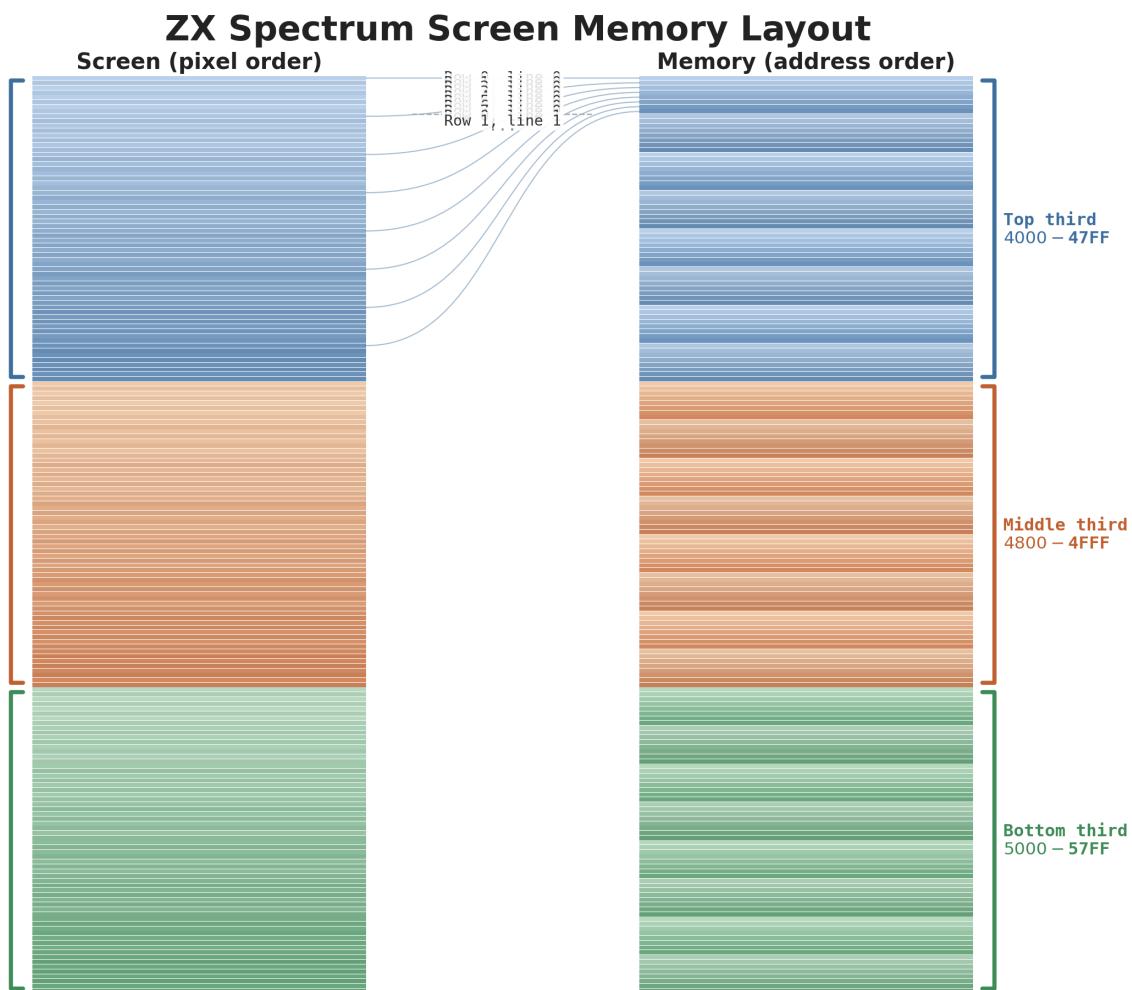
\$5800 - \$5AFF Attributes 768 bytes (32 x 24 colour cells)

Пиксельная область хранит растр: 256 пикселей по горизонтали, упакованные по 8 в байт, что даёт 32 байта на строку. При 192 строках это $32 \times 192 = 6\,144$ байта. Каждый байт представляет 8 горизонтальных пикселей, где бит 7 — крайний левый, а бит 0 — крайний правый.

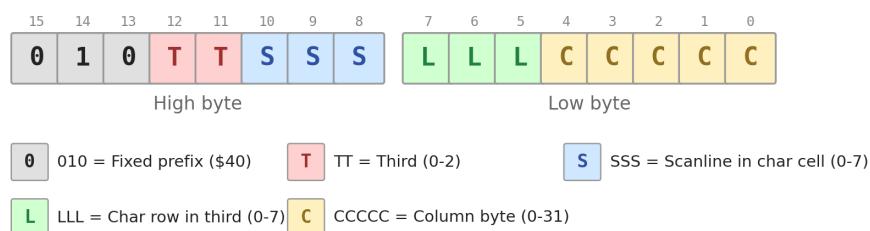
Область атрибутов хранит цветовую информацию: один байт на 8x8 знакоместо. 32 столбца и 24 строки, итого $32 \times 24 = 768$ байт.

Вместе: $6\,144 + 768 = 6\,912$ байт. Это весь экран.

Пиксельные данные и данные атрибутов служат разным целям, но тесно связаны. Каждый пиксельный байт управляет 8 точками на экране; байт атрибута соответствующего 8x8 знакоместа определяет, каким цветом эти точки отображаются. Измени пиксель — изменишь форму. Измени атрибут — изменишь цвет. Но цвет можно менять только для целого блока 8x8 — не попиксельно. Это



Address Bit Layout: 010TTSSS LLLCCCCC



Example: Third 1, scanline 3, char row 5, column 10

High: 010 | 01 | 011 = \$4B (TT=01, SSS=011)

Low: 101 | 01010 = \$AA (LLL=101, CCCCC=01010)

Address: \$4BAA → pixel row 107 (third 1, row 5, line 3), column 10

Рис. 4: ZX Spectrum screen memory layout with thirds, character cells, and attribute area

и есть «конфликт атрибутов», определяющий визуальный характер Spectrum, и мы вернёмся к нему чуть позже.

Но сначала — головоломка: почему пиксельные строки перемешаны?

Чересстрочность: Где живут строки

Если бы Spectrum хранил пиксельные строки последовательно, строка 0 была бы по адресу \$4000, строка 1 — по \$4020, строка 2 — по \$4040, и так далее. Каждая строка — 32 байта, значит строка N просто по $\$4000 + N * 32$. Просто, быстро, разумно.

Но не так всё устроено.

Экран разделён на три **трети**, каждая высотой 64 пиксельных строки. Внутри каждой трети строки чередуются по знакоместам. Вот где реально находятся первые 16 строк:

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

Посмотри на паттерн. Первые 8 строк — это 8 развёрточных строк знакоместа 0, но они расположены через 256 байт, а не через 32. Внутри этих 8 строк старший байт адреса увеличивается на 1 с каждой: \$40, \$41, \$42, ... \$47. Затем строка 8 перескакивает на \$4020 — назад к старшему байту \$40, но с младшим байтом, увеличенным на 32.

Вот полная картина для верхней трети экрана:

```
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
```

Средняя третья начинается с \$4800 и следует тому же паттерну. Нижняя третья начинается с \$5000.

Почему?

Причина — ULA, Uncommitted Logic Array, формирующая видеосигнал. ULA считывает один байт пиксельных данных и один байт атрибутов для каждого 8-пиксельного знакоместа при развёртке. Ей нужны оба байта в определённые моменты, когда луч проходит по экрану.

Чересстрочная раскладка означала, что логика формирования адресов в ULA могла быть построена с меньшим числом вентилей. Когда ULA сканирует слева направо по строке знакомест, она инкрементирует младшие 5 бит адреса (столбец). Дойдя до правого края, она инкрементирует старший байт для перехода к следующей развёрточной строке внутри того же знакоместа. Завершив все 8 строк, она сбрасывает старший байт и продвигает биты строки в младшем байте.

Это элегантно с аппаратной точки зрения. Формирование адресов ULA — это простая комбинация счётчиков: никакого умножения, никакой сложной адресной арифметики. Разводка печатной платы была проще, количество вентилей меньше, а микросхема дешевле в производстве.

Программист платит эту цену.

Битовая раскладка: Декодирование (x, y) в адрес

Чтобы точно понять чересстрочность, посмотри, как координата Y отображается в 16-битный экранный адрес. Рассмотрим пиксель в столбце x (0-255) и строке y (0-191). Байт, содержащий этот пиксель, находится по адресу:

Low byte: L L L C C C C C

Где: - TT = какая треть экрана (0, 1 или 2). Биты 7-6 координаты y. - SSS = развёрточная строка внутри знакоместа (0-7). Биты 2-0 координаты y. - LLL = номер строки знакомест внутри трети (0-7). Биты 5-3 координаты y. - CCCCC = столбец в байтах (0-31). Это x / 8, или эквивалентно биты 7-3 координаты x.

Ключевой момент: биты y расположены не по порядку. Биты 7-6 идут в одно место, биты 5-3 — в другое, а биты 2-0 — в третье. Координата y нарезана и распределена по адресу.

Визуализируем на конкретном примере. Пиксель (80, 100):

```
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)
```

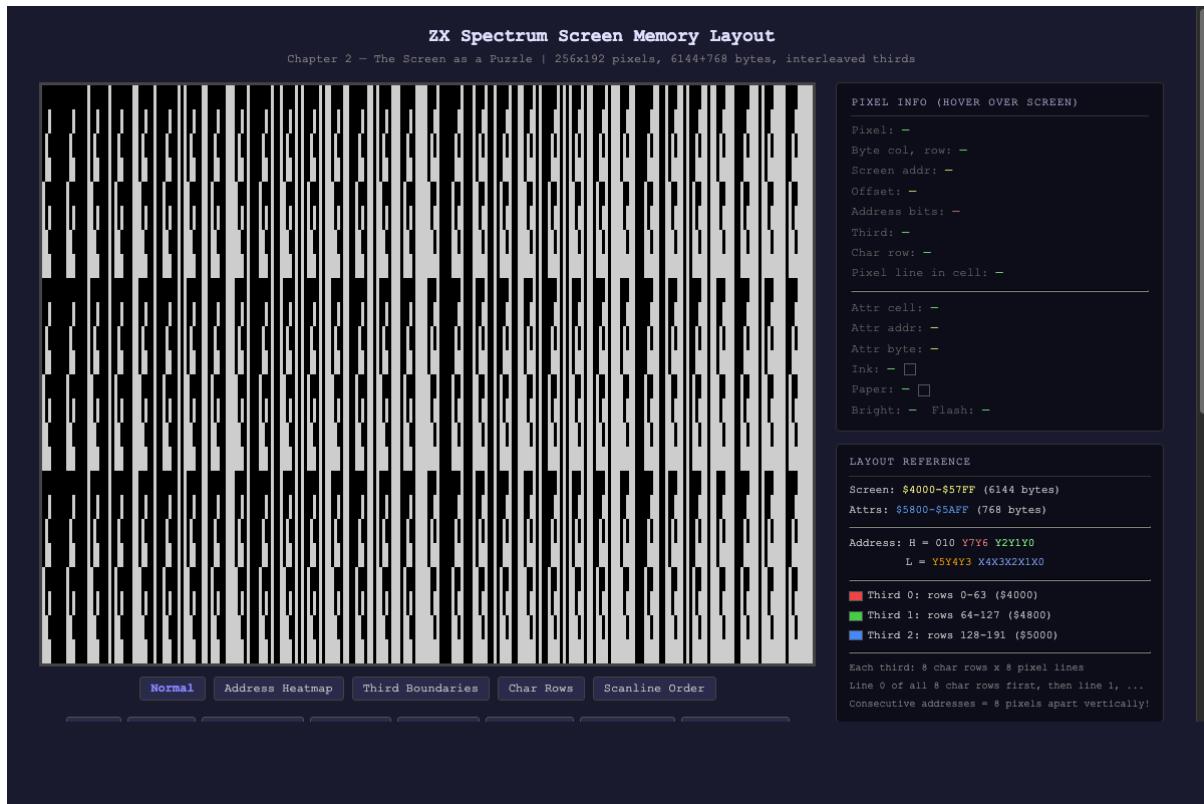


Рис. 5: Раскладка экранной памяти ZX Spectrum — чересстрочные трети с цветовой кодировкой битовых полей адреса

High byte: 0 1 0 0 1 1 1 0 0 = \$4C
Low byte: 1 0 0 0 1 0 1 0 = \$8A

Address: \$4C8A

Бит внутри этого байта определяется тремя младшими битами х. Бит 7 — крайний левый пиксель, поэтому позиция пикселя (x AND 7) соответствует биту 7 - (x AND 7).

Вычисление адреса на Z80

Преобразование (x, y) в экранный адрес нужно делать быстро и часто. Вот стандартная процедура:

```
; 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 такт — не дёшево. В тесном внутреннем цикле, обрабатывающем тысячи пикселей, ты не будешь вызывать эту процедуру для каждого пикселя. Вместо этого вычисляешь начальный адрес один раз, а затем перемещаешься по экрану с помощью быстрых манипуляций указателем — что подводит нас к самой важной процедуре в графическом программировании на Spectrum.

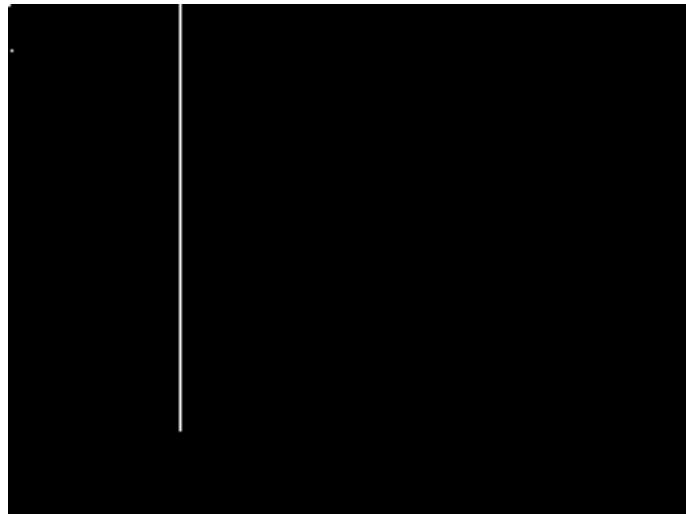


Рис. 6: Демо построения пикселей — отдельные пиксели, размещённые на экране с помощью процедуры вычисления адреса

DOWN_HL: Перемещение на одну пиксельную строку вниз

У тебя есть указатель в HL на некоторый байт на экране. Ты хочешь переместить его на одну пиксельную строку вниз — к байту в том же столбце, но на

одну развёрточную строку ниже. Что тут сложного?

На линейном фреймбуфере добавляешь 32 (число байт в строке). Одна ADD HL, DE при DE = 32: 11 тактов, готово.

На Spectrum это головоломка внутри головоломки. Перемещение на одну пиксельную строку вниз означает:

1. **Внутри знакоместа** (развёрточные строки 0–6 к 1–7): инкремент H. Биты развёрточной строки находятся в младших 3 битах H, поэтому INC H перемещает на одну строку вниз.
2. **При переходе через границу знакоместа** (строка 7 к строке 0 следующего знакоместа): сбросить биты развёрточной строки в H обратно в 0 и добавить 32 к L для перехода к следующей строке знакомест.
3. **При переходе через границу трети** (нижняя строка знакоместа 7 одной трети к верхней строке знакоместа 0 следующей): сбросить L и добавить 8 к H для перехода к следующей трети. Эквивалентно — добавить \$0800 к адресу.

Классическая процедура обрабатывает все три случая:

```
; 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
```

Эта процедура занимает разное время в зависимости от случая:

Случай	Частота	Такты
Внутри знакоместа	7 из 8 строк	$4 + 4 + 7 + 11 = \mathbf{26}$
Граница знакоместа, та же треть	7 из 64 строк	$4 + 4 + 7 + 5 + 4 + 7 + 4 + 5 + 4 + 7 + 4 + 10 = \mathbf{65}$

Случай	Частота	Такты
Граница трети	2 из 192 строк	$4 + 4 + 7 + 5 + 4 + 7 + 4 + 11 = \mathbf{46}$

Типичный случай — внутри знакоместа — быстрый: 26 тактов (T-state) (условный RET, который срабатывает, стоит 11T, а не 5T). Редкий случай (переход через строку знакомест внутри той же трети) — 65 тактов (T-state). В среднем по всем 192 строкам стоимость составляет около **30,5 такта (T-state) на вызов**.

Это среднее скрывает проблему. Если ты итерируешь по всему экрану сверху вниз, вызывая DOWN_HL на каждой строке, те редкие вызовы по 65 тактов (T-state) дают непредсказуемые пики во времени. Для демо-эффекта, которому нужен стабильный тайминг на каждой строке развёртки, эта нестабильность неприемлема.

Оптимизация Introspec'a

В декабре 2020 года Introspec (spke) опубликовал детальный анализ на Hure под названием «Ещё раз про DOWN_HL». Статья рассматривала задачу эффективной итерации по всему экрану сверху вниз — не стоимость одного вызова, а полную стоимость прохода HL через все 192 строки.

Наивный подход — вызов классической процедуры DOWN_HL 191 раз — стоит **5 825 тактов (T-state)** для полного прохода по экрану. Цель Introspec'a была найти самый быстрый способ пройти все 192 строки, посещая каждый экранный адрес в порядке сверху вниз.

Его ключевой идеей было использование **раздельных счётчиков**. Вместо проверки битов адреса после каждого инкремента для обнаружения пересечения границ он структурировал цикл в соответствии с трёхуровневой иерархией экрана:

```
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
```

Внутренняя операция — просто INC H — 4 такта. Никаких проверок, никаких переходов. Переходы между строками знакомест и третями происходят в фиксированных, предсказуемых точках цикла, поэтому во внутреннем цикле нет условной логики вообще.

Результат: **2 343 такта** для полного прохода по экрану. Это улучшение на 60% по сравнению с классическим подходом, и стоимость на строку абсолютно предсказуема — никакой нестабильности.

Была также элегантная вариация, приписываемая RST7, с двойным счётчиком, где внешний цикл поддерживает пару счётчиков, естественно отслеживающих границы знакомест и третей. Тело внутреннего цикла сводится к одному INC H,

а обработка границ вложена в манипуляции со счётчиками на уровне внешнего цикла.

Практический урок: когда нужно итерировать по экрану Spectrum по порядку, не вызывай общую процедуру DOWN_HL 191 раз. Перестрой цикл в соответствии с естественной иерархией экрана, и ветвления исчезнут.

Вот упрощённая версия подхода с раздельными счётчиками:

```
; 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

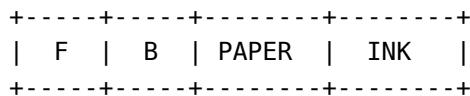
    dec  c                  ; 4T
    jr    nz, .third_loop     ; 12T/7T
```

Директива REPT 7 (поддерживается sjasmplus) повторяет блок 7 раз на этапе ассемблирования — частичная развёртка. Внутри этого блока перемещение на одну развёрточную строку вниз — это один INC H. Никаких проверок, никаких переходов. Продвижение по строкам знакомест и по третям происходит на фиксированных внешних границах цикла.

Память атрибутов: 768 байт, которые изменили всё

Под пиксельными данными, по адресам \$5800–\$5AFF, находится память атрибутов. Это 768 байт — по одному на каждое 8x8 знакоместо на экране, расположенных последовательно слева направо, сверху вниз. В отличие от пиксельной области, раскладка атрибутов полностью линейна: ячейка (col, row) находится по адресу $\$5800 + \text{row} * 32 + \text{col}$.

Каждый байт атрибута имеет следующую структуру:



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)

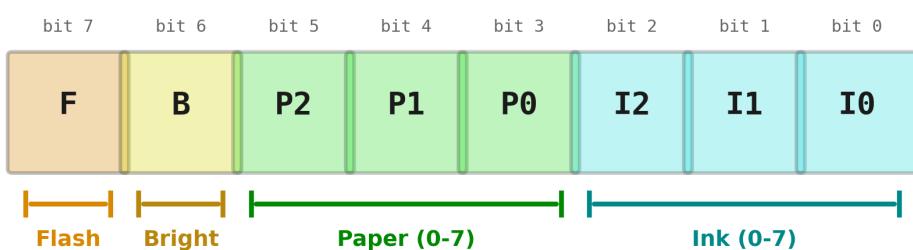
3-битные коды цветов:

1 = Blue	5 = Cyan
2 = Red	6 = Yellow
3 = Magenta	7 = White

С битом BRIGHT каждый цвет имеет обычный и яркий вариант. Чёрный остаётся чёрным с яркостью или без, так что всего палитра — 15 различных цветов:

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

ZX Spectrum Attribute Byte Layout



Example:

\$41 = 01000001

= BRIGHT Blue ink on Black paper



Рис. 7: Attribute byte bit layout showing flash, bright, paper, and ink fields

Байт атрибута $\$47 = 01000111$: мерцание (flash) выкл (бит 7 = 0), яркость (bright) вкл (бит 6 = 1), фон (paper) = 000 (чёрный), чернила (ink) = 111 (белый). Яркий белый текст на чёрном фоне. Неяркая версия — $\$07 = 00000111$ — умолчание Spectrum после BORDER 0: PAPER 0: INK 7.

Такие битовые детали важны, когда ты конструируешь значения атрибутов на скорости. Типичный паттерн:

```
; 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
```

Конфликт атрибутов

Вот определяющее ограничение ZX Spectrum: внутри каждого 8x8 пиксельного знакоместа может быть только **два цвета** — ink и paper. Каждый установленный пиксель (1) отображается цветом ink. Каждый сброшенный пиксель (0) отображается цветом paper. Нельзя иметь три цвета, или градиенты, или по-пиксельную раскраску внутри одного знакоместа.

Это означает, что если красный спрайт перекрывается с зелёным фоном, знакоместо 8x8, содержащее перекрытие, должно выбрать: все установленные пиксели в этой ячейке — либо красные, либо зелёные. Нельзя иметь часть красных и часть зелёных установленных пикселей в одной ячейке. Визуальный результат — режущий глаз блок цвета, «конфликтующий» с окружением — печально известный конфликт атрибутов.

+-----+-----+	
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.

Многие ранние игры для Spectrum просто избегали проблемы: монохромная графика или персонажи, тщательно выровненные по сетке 8x8. Игры вроде Knight Lore и Head Over Heels использовали одну пару ink/paper для всей игровой области, полностью устраняя конфликт цвета.

Но демосцена увидела это иначе. Конфликт атрибутов — это не просто ограничение, это **творческое ограничение**. Сетка 8x8 диктует определённую эстетику: смелые блоки цвета, чёткие геометрические паттерны, намеренное

использование контраста. Демо-эффекты, работающие целиком в пространстве атрибутов — туннели, плазмы, скроллеры — могут обновлять 768 байт за кадр вместо 6 144, освобождая огромное количество тактов для вычислений. Когда весь экран управляется атрибутами, конфликт становится неактуальным, потому что ты не смешиваешь спрайты с фонами — атрибуты *и есть* графика.

Демо Introspec'a Eager (2015) полностью построило свой визуальный язык на этом понимании. Эффект туннеля, хаотический зумер и анимация цветового цикла — все работают с атрибутами, не с пикселями. Результат — эффект, работающий на полной частоте кадров с запасом для цифровых барабанов и изощрённого скриптового движка. Конфликт не проблема, потому что ограничение было принято с самого начала.

Бордюр: Больше, чем декорация

Область отображения 256x192 пикселей расположена в центре экрана, окружённая широким бордюром. Цвет бордюра устанавливается записью в порт \$FE:

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

На цвет бордюра влияют только биты 0-2 записываемого байта. Есть 8 цветов (0-7), без ярких вариантов — палитра бордюра не-яркая. Биты 3 и 4 порта \$FE управляют выходами MIC и EAR (магнитофонный интерфейс и бипер), поэтому их нужно маскировать или устанавливать надлежащим образом, если ты не намерен шуметь.

Смена цвета бордюра вступает в силу немедленно — на следующей же отрисовываемой строке развёртки. Именно это делает бордюр столь полезным инструментом отладки. Как мы видели в главе 1, смена цвета бордюра до и после секции кода создаёт видимую полосу, высота которой показывает стоимость кода в тактах. Бордюр — твой осциллограф.

Эффекты в бордюре

Поскольку смена цвета бордюра видна на следующей строке развёртки, точно рассчитанные по времени инструкции OUT могут создавать многоцветные полосы, растровые бары и даже грубую графику в области бордюра.

Базовый принцип: ULA рисует одну строку развёртки каждые 224 такта (на Pentagon). Если выполнить инструкцию OUT (\$FE), А в нужный момент, ты изменишь цвет бордюра в определённой горизонтальной позиции текущей строки. Выполняя быструю последовательность инструкций OUT с разными значениями цвета, можно рисовать горизонтальные цветные полосы в бордюре.

```
; 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

```

Более продвинутые бордюрные эффекты могут создавать градиентные полосы, бегущий текст или даже низкоразрешающие изображения. Сложность экстремальна: у тебя 224 такта на строку, и каждая смена цвета стоит минимум 18 тактов (7 на LD A,n + 11 на OUT). Это даёт примерно 12 смен цвета на строку, что означает максимум 12 горизонтальных цветных полос на линию.

Демо-кодеры довели это до поразительных крайностей. Предзагружая несколько регистров значениями цветов и используя более быстрые последовательности вроде OUT (C), А с последующими обменами регистров, они выжимают больше смен цвета на строку. Бордюр становится самостоятельным дисплеем — холстом за пределами холста.

Для наших целей главная роль бордюра — та же, что в главе 1: бесплатный, всегда доступный визуализатор тайминга. Когда ты будешь оптимизировать процедуру заполнения экрана далее в этой главе, бордюр покажет твой прогресс.

Практика: Заполнение шахматным узором

Пример в `chapters/ch02-screen-as-puzzle/examples/fill_screen.a80` заполняет пиксельную область шахматным узором, а атрибуты — ярким белым на синем. Разберём его по секциям.

```

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)

```

Код размещён по адресу \$8000 — безопасно в неспорной памяти на всех моделях Spectrum. Константы именуют ключевые адреса и размеры.

```

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

```

Здесь использован классический трюк самокопирования LDIR. Записывается \$55 (двоичное 01010101) в первый байт по \$4000, затем копируется из каждого байта в следующий для 6 143 байт. Результат: каждый байт пиксельной области содержит \$55, что даёт чередующиеся установленные/брошенные пиксели — шахматную доску. Поскольку паттерн одинаков в каждом байте,

чересстрочный порядок строк не имеет значения — каждая строка получает один и тот же паттерн.

Стоимость: LDIR копирует 6 143 байта. Последняя итерация стоит 16T, все остальные — 21T: $(6\ 143 - 1) \times 21 + 16 = 128\ 998$ тактов (T-state). Почти два полных кадра на Pentagon. Это нормально для однократной настройки, но ты ни за что не станешь так делать в цикле покадровой отрисовки.

```
; --- 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
```

Тот же приём для атрибутов. Значение \$4F декодируется как: flash выкл (0), bright вкл (1), paper синий (001), ink белый (111). Каждое 8x8 знакоместо получает яркие белые чернила на синей бумаге. Пиксели шахматной доски — установленные/брошенные, поэтому ты видишь чередующиеся белые и синие точки — классический визуальный паттерн ZX Spectrum.

Стоимость: LDIR копирует 767 байт — $(767 - 1) \times 21 + 16 = 16\ 102$ такта (T-state).

```
; --- Border: blue ---
ld    a, 1
out   ($FE), a

; Infinite loop
.wait:
halt
jr    .wait
```

Устанавливает бордюр в синий (цвет 1) в тон бумаге, создавая визуально чистую рамку. Затем бесконечный цикл с HALT между кадрами. HALT ожидает следующего маскируемого прерывания, которое срабатывает раз в кадр — это пульс покоя каждой программы для Spectrum.

Что попробовать

Загрузи `fill_screen.a80` в ассемблер и эмулятор. Затем экспериментируй:

- Замени \$55 на \$AA для обратной шахматной доски, или \$FF для сплошной заливки, или \$81 для вертикальных полос.
- Замени \$4F на \$07, чтобы увидеть тот же паттерн без BRIGHT, или на \$38 для белой бумаги с чёрными чернилами (инверсия умолчания).
- Попробуй \$C7 — это устанавливает бит flash. Наблюдай, как знакоместа чередуют цвета ink и paper с частотой ~1,6 Гц.
- Замени пиксельную заливку LDIR на цикл с DOWN_HL, записывающий разные паттерны в разные строки. Теперь ты увидишь чересстрочность в действии: если записать \$FF в строки 0-7 (развёрточные строки первого знакоместа), заполненная область будет выглядеть как 8 горизонтальных полосок, разделённых промежутками — потому что эти строки отстоят на 256 байт, а не на 32.



Рис. 8: Заливка экрана чередующимися байтами — шахматный узор яркобелым на синем

Навигация по экрану: Практическая сводка

Вот основные операции с указателями для экрана Spectrum, собранные в одном месте. Это строительные блоки каждой графической процедуры.

Перемещение вправо на один байт (8 пикселей)

```
inc l           ; 4T
```

Это работает внутри строки знакомест, потому что столбец находится в младших 5 битах L. Если нужно пересечь границу байтов на правом краю (столбец 31 к столбцу 0 следующей строки), потребуется полный DOWN_HL плюс сброс L — но обычно это не нужно, потому что циклы имеют ширину 32 байта.

Перемещение вниз на одну пиксельную строку

```
inc h           ; 4T      (within a character cell)
```

Это работает для 7 из 8 строк. На 8-й строке нужна полная логика пересечения границы из процедуры DOWN_HL выше.

Перемещение вниз на одну строку знакомест (8 пикселей)

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

Это продвигает на одну строку знакомест внутри трети. Если L переполняется (флаг переноса установлен), ты пересёк границу следующей трети и нужно добавить 8 к H.

Перемещение вверх на одну пиксельную строку

```
dec h           ; 4T      (within a character cell)
```

Обратная операция INC H. Те же проблемы с границами знакомест и третей. Вот полная процедура UP_HL, зеркальная к DOWN_HL:

```
; 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
```

Есть тонкая оптимизация, предложенная Art-top (Артём Топчий): замена and 7 / cp 7 на cpl / and 7. После DEC H, если младшие 3 бита H обернулись с 000 на 111, мы пересекли границу знакоместа. Классический тест проверяет AND 7 и затем сравнивает с 7. Оптимизированная версия сначала инвертирует: если биты равны 111, CPL делает их 000, и AND 7 даёт ноль. Это экономит 1 байт и 3 такта (T-state) на пути пересечения границы:

```
; 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

```

Тот же трюк с CPL / AND 7 работает и в DOWN_HL, хотя условие границы там проверяет 000 (которые CPL превращает в 111, тоже ненулевые после AND), поэтому при движении вниз он не помогает. Именно направление *вверх* — то, где классическому коду нужен дополнительный СР 7, который оптимизация устраняет.

Вычисление адреса атрибута по адресу пикселя

Если HL указывает на байт в пиксельной области, можно вычислить соответствующий адрес атрибута. Вспомни структуру пиксельного адреса: H = 010TTSSS, L = LLLCCCCC. Адрес атрибута для той же ячейки атрибутов — \$5800 + TT * 256 + LLL * 32 + CCCCC. Поскольку L уже содержит LLL * 32 + CCCCC (диапазон 0-255), адрес атрибута — просто (\$58 + TT) : L. Всё, что нужно сделать — извлечь два бита TT из H, объединить их с \$58 и оставить L без изменений:

```

; 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

```

Каждая техника в остальной части книги определяется раскладкой экрана, описанной в этой главе. Вот почему каждый элемент важен:

Особый случай: когда в H биты строки развёртки = 111. Если ты итерируешь по ячейке атрибутов сверху вниз и только что обработал последнюю строку развёртки (строка развёртки 7), младшие 3 бита H равны 111. В этом случае существует более быстрое преобразование из 4 инструкций, предложенное Art-top:

```

; 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

```

Скроллинг (глава 17) — это где раскладка больше всего мешает. Прокрутка экрана вверх на один пиксель означает перемещение 32 байт каждой строки по адресу строки выше неё. На линейном фреймбуфере это одно большое блочное копирование. На Spectrum адреса источника и назначения для каждой строки связаны логикой DOWN_HL — не фиксированным смещением. Процедура прокрутки должна навигировать по чересстрочности для каждой копируемой строки.

Врезка: Agon Light 2

Дисплей Agon Light 2 управляется VDP (Video Display Processor) — микроконтроллером ESP32, на котором работает библиотека FabGL. ЦП eZ80 обменивается с VDP через последовательный канал, отправляя команды для установки графических режимов, рисования пикселей, определения спрайтов и управления палитрами.

Здесь нет чересстрочной раскладки экрана. Нет конфликта атрибутов. VDP поддерживает несколько раstroвых режимов с различными разрешениями (от 640x480 до 320x240 и ниже), с 64 цветами или полными RGBA-палитрами в зависимости от режима. Аппаратные спрайты (до 256) и тайловые карты поддерживаются нативно.

Что меняется для программиста:

- **Никакой адресной головоломки.** Пиксельные координаты линейно отображаются на позиции в буфере. Тебе не нужны DOWN_HL или обход экрана с раздельными счётчиками.
- **Никакого конфликта атрибутов.** Каждый пиксель может быть любого цвета. Ограничение сетки 8x8 не существует.
- **Нет прямого доступа к фреймбуферу.** ЦП не может напрямую писать в видеопамять так, как ЦП Spectrum пишет в \$4000. Вместо этого ты отправляешь команды VDP через последовательный канал. Нарисовать пиксель — значит отправить последовательность команд, а не записать байт. Это вносит задержку — последовательный канал работает на 1 152 000 бод — но зато ЦП свободен во время рендеринга.
- **Нет трюков на уровне тактов с бордюром.** VDP управляет таймингом отображения независимо. Ты не можешь создавать растроевые эффекты, синхронизируя инструкции OUT, потому что конвейер отображения отвязан от тактовой частоты ЦП.

Для программиста Spectrum Agon одновременно освобождает и разочаровывает. Ограничения, которые порождали творческие решения на Spectrum, попросту не существуют — но не существуют и прямые аппаратные трюки, которые эти ограничения делали возможными. Ты меняешь головоломку на API.

- 6 912-байтовый экран Spectrum состоит из **6 144 байт пиксельных данных** по адресам \$4000–\$57FF и **768 байт атрибутов** по адресам \$5800–\$5AFF.
- Пиксельные строки **переворачиваются** по знакоместам: адрес кодирует у как 010 TT SSS (старший байт) и LLL CCCCC (младший байт), где биты у перемешаны по адресу.

- Перемещение **на одну пиксельную строку вниз** внутри знакоместа — это просто INC H (4 такта). Пересечение границ знакомест и третей требует дополнительной логики.
 - Классическая процедура **DOWN_HL** обрабатывает все случаи, но стоит до 77 тактов на границах. Для полноэкранной итерации **циклы с раздельными счётчиками** (подход Introspec'a) снижают общую стоимость на 60% и устраняют нестабильность тайминга.
 - Каждый байт атрибута кодирует **Flash, Bright, Paper** и **Ink** в формате FBPPPII. Только **два цвета на 8x8 знакоместо** — это конфликт атрибутов.
 - Конфликт атрибутов — это не просто ограничение, а **творческое ограничение**, определившее визуальную эстетику Spectrum и приведшее к эффективным демо-эффектам, работающим только с атрибутами.
 - Цвет **бордюра** устанавливается через OUT (\$FE), A (биты 0-2), и изменения видны на следующей строке развёртки, что делает его **инструментом отладки тайминга** и холстом для демосценовых растровых эффектов.
 - **Agon Light 2** не имеет чересстрочной раскладки, конфликта атрибутов и прямого доступа к фреймбуферу — он заменяет головоломку на командный API VDP.
-

Попробуй сам

- Вычисли адреса.** Возьми 10 случайных координат (x, y) и рассчитай экранный адрес вручную, используя битовую раскладку 010TTSSS LLLCCCCC. Затем напиши маленькую процедуру на Z80, которая ставит один пиксель по каждой координате, и проверь, что твои вычисления совпадают.
 - Визуализируй чересстрочность.** Модифицируй fill_screen.a80, чтобы записать разные значения в первые 8 строк. Запиши \$FF (сплошной) в строку 0 и \$00 (пустой) в строки 1-7. Поскольку строки 0-7 находятся по адресам \$4000, \$4100, ..., \$4700, тебе нужно менять H для достижения каждой строки. Результат должен быть одной яркой линией вверху с промежутком из 7 пустых линий до следующей сплошной линии на строке 8.
 - Замерь DOWN_HL.** Используй тестовую обвязку с цветом бордюра из главы 1. Вызови классическую процедуру DOWN_HL 191 раз (для полного прохода по экрану) и измерь полосу. Затем реализуй версию с раздельными счётчиками и сравни. Версия с раздельными счётчиками должна дать заметно более короткую полосу.
-
- Полосы бордюра.** После HALT выполнни тесный цикл, меняющий цвет бордюра на каждой строке развёртки в течение 64 строк. Используй 8 цветов бордюра по порядку (0, 1, 2, 3, 4, 5, 6, 7, повтор). Ты увидишь горизонтальные радужные полосы в верхнем бордюре. Подбери задержку между инструкциями OUT, пока полосы не станут чистыми и стабильными.
 - Дисплей Spectrum на 6 912 байт состоит из **6 144 байт пиксельных данных** по адресам \$4000-\$57FF и **768 байт атрибутов** по адресам \$5800-\$5AFF.
 - Строки пикселей **чересстрочно перемешаны** по знакоместам: адрес кодирует у как 010 TT SSS (старший байт) и LLL CCCCC (младший байт), где

биты у перетасованы по адресу.

- Перемещение **на одну строку пикселей вниз** внутри знакоместа — это просто INC H (4 такта (T-state)). Пересечение границ знакомест и третей требует дополнительной логики.
 - Классическая процедура **DOWN_HL** обрабатывает все случаи, но стоит до 65 тактов (T-state) на границах. Для полноэкранной итерации **циклы с раздельными счётчиками** (подход Introspec'a) снижают общую стоимость на 60% и устраняют нестабильность тайминга.
 - Каждый байт атрибута кодирует **мерцание (flash), яркость (bright), фон (paper) и чернила (ink)** в формате FBPPPIII. Только **два цвета на ячейку 8x8** — это и есть конфликт атрибутов.
 - Конфликт атрибутов — это не просто ограничение, а **творческое ограничение**, которое определило визуальную эстетику Spectrum и привело к эффективным демо-эффектам, работающим только с атрибутами.
 - Цвет **бордюра** задаётся через OUT (\$FE), A (биты 0–2), и изменения видны на следующей строке разёртки, что делает его **инструментом отладки тайминга** и холстом для растровых эффектов демосцены.
 - **Agon Light 2** не имеет чересстрочной раскладки, конфликта атрибутов и прямого доступа к фреймбуферу — он заменяет головоломку на командный API через VDP.
-

Далее: Глава 3 — Инструментарий демосценера. Развёрнутые циклы, самодифицирующийся код, стек как канал данных и техники, позволяющие делать невозможное в рамках бюджета.

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.

Источники: Introspec «Ещё раз про DOWN_HL» (Hype, 2020); Introspec «GO WEST Part 1» (Hype, 2015) об эффектах спорной памяти по экранным адресам; Introspec «Making of Eager» (Hype, 2015) о проектировании эффектов на основе атрибутов; документация ULA Spectrum о логике раскладки памяти; Art-top (личное общение, 2026) об оптимизированных UP_HL и быстром преобразовании пиксельного адреса в адрес атрибута.

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.

Глава 3: Инструментарий демосценера

В каждом ремесле есть свой набор приёмов — паттернов, к которым мастера тянутся настолько инстинктивно, что перестают воспринимать их как приёмы. Демосценер на Z80 тянеться к техникам из этой главы.

Эти паттерны — развернутые циклы, самомодифицирующийся код, стек как канал данных, LDI-цепочки, генерация кода и RET-цепочки — встречаются практически в каждом эффекте, который мы будем строить во второй части. Именно они отличают демо, укладывающееся в один кадр, от демо, которому нужно три. Освой их здесь — и ты будешь узнавать их повсюду.

Развёрнутые циклы и самомодифицирующийся код

Стоимость цикла

Рассмотрим простейший внутренний цикл: заполнение нулями 256 байт памяти.

```
; 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)
```

Каждая итерация стоит $7 + 6 + 13 = 26$ тактов на запись одного байта. Из них лишь 7 тактов делают полезную работу — остальное накладные расходы. Это 73% впустую. На 256 байт: $256 \times 26 - 5 = 6\,651$ такт. На машине с бюджетом 71 680 тактов на кадр эти потерянные такты ощутимы.

Развёртка: размен RAM на скорость

Решение жёсткое и эффективное: выписать тело цикла N раз и убрать цикл.

```
; 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

```

Теперь каждый байт стоит $7 + 6 = 13$ тактов. Нет DJNZ. Нет счётчика. Итого: $256 \times 13 = 3\,328$ тактов — вдвое меньше цикловой версии.

Цена — размер кода: 256 повторений занимают 512 байт против 7 у цикла. Ты размениваешь RAM на скорость.

Когда развёртывать: Внутренние циклы, выполняющиеся тысячи раз за кадр — очистка экрана, отрисовка спрайтов, копирование данных.

Когда НЕ развёртывать: Внешние циклы, выполняющиеся один-два раза за кадр. Экономия 5 тактов на 24 итерациях даёт 120 тактов — меньше трёх NOP. Не стоит раздувания.

Практический компромисс — *частичная развёртка*: развернуть 8 или 16 итераций внутри цикла, оставить DJNZ для внешнего счётчика. Пример push_fill.a80 в каталоге examples/ этой главы делает именно это: 16 PUSH за итерацию, 192 итерации.

Самомодифицирующийся код: секретное оружие Z80

У Z80 нет кэша инструкций, нет буфера предвыборки, нет конвейера. Когда процессор считывает байт инструкции из ОЗУ, он читает то, что там находится *прямо сейчас*. Если ты изменил этот байт один цикл назад, процессор увидит новое значение. Это гарантированное свойство архитектуры.

Самомодифицирующийся код (SMC) означает запись в байты инструкций во время выполнения. Классический паттерн — подмена непосредственного операнда:

```

; 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
    ;

```

Инструкция `ld (patch + 1), a` записывает в непосредственный операнд следующей `ld (hl), $00`, превращая её в `ld (hl), $AA` или что угодно другое. Процессор выполняет те байты, которые находит. Вот несколько распространённых паттернов SMC:

Подмена опкодов. Можно заменить саму инструкцию. Нужен цикл, который иногда инкрементирует HL, а иногда декрементирует? Перед циклом запиши опкод INC HL (\$23) или DEC HL (\$2B) в байт инструкции. Внутри внутреннего цикла вообще нет ветвления — нужная инструкция уже на месте. Сравни с подходом «ветвление на каждую итерацию», который стоил бы 12 тактов (JR NZ) на каждый пиксель.

Сохранение и восстановление указателя стека. Этот паттерн встречается постоянно при использовании PUSH-трюков (ниже):

```
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
```

Инструкция `ld (nn), sp` сохраняет текущий SP прямо в операнд последующей `ld sp, nn`. Никаких временных переменных. Это идиоматический код демосцены на Z80.

Самомодифицирующиеся переменные: паттерн \$+1

Самый распространённый паттерн SMC на ZX Spectrum — это не подмена опкодов и не сохранение SP, а встраивание *переменной* непосредственно в непосредственный операнд инструкции. Идея проста: вместо того чтобы хранить счётчик в именованной ячейке памяти и загружать его через `LD A, (nn)` за 13 тактов, ты позволяешь байту операнда самой инструкции *быть переменной*.

```
.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
```

`ld a, 0` извлекает свой операнд как часть обычного декодирования инструкции — всего 7 тактов, и значение уже в А. Сравни с загрузкой из отдельного адреса памяти: `ld a, (counter)` стоит 13 тактов, плюс тебе всё равно нужен отдельный `ld (counter)`, а за 13 тактов для обратной записи. Версия с SMC читает переменную бесплатно (это часть выборки инструкции) и платит 13 тактов лишь однажды — за обратную запись.

В sjasmplus можно поставить метку на `$+1`, чтобы дать встроенной переменной читаемое имя:

```
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
```

Этот паттерн встречается повсюду в коде для ZX Spectrum: позиции скроллинга, счётчики кадров анимации, аккумуляторы фаз эффектов, флаги направления. Любое однобайтовое значение, которое сохраняется между вызовами, — кандидат. Ты будешь постоянно видеть его в частях II и V — практически каждая процедура эффекта в этой книге использует как минимум одну самомодифицирующуюся переменную.

Принято именовать такие метки с префиксом `.smc_` или размещать их сразу после инструкции, которую они модифицируют. В любом случае намерение должно быть понятно каждому, кто читает исходный код. Как мы отмечали в главе 2, локальные метки (`.label`) предотвращают коллизии имён, когда несколько подпрограмм имеют собственные встроенные переменные.

Предупреждение. SMC безопасен на Z80, eZ80 и любом клоне Spectrum. Он *не безопасен* на современных процессорах с кэшем (x86, ARM) без явного

сброса кэша инструкций. Если будешь портировать на другую архитектуру, это первое, что сломается.

Стек как канал данных

Почему PUSH — самая быстрая запись на Z80

Инструкция PUSH записывает 2 байта в память и декрементирует SP, всё за 11 тактов. Сравним альтернативы записи данных по экранному адресу:

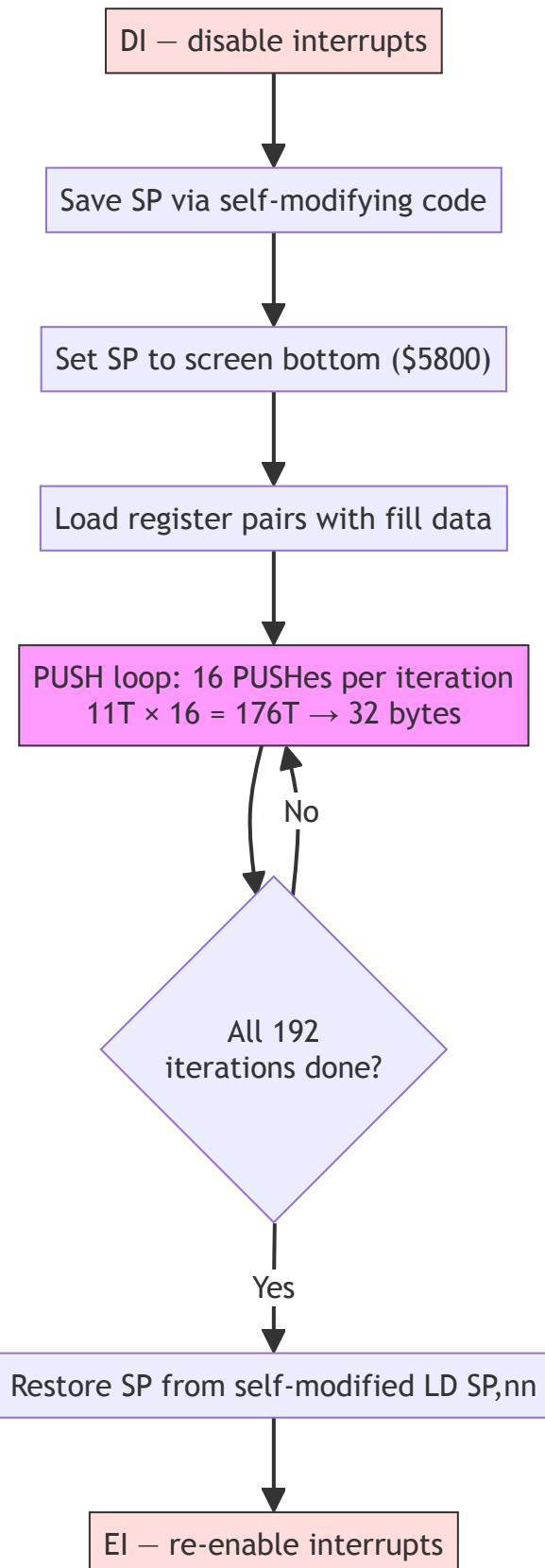
Метод	Записано байт	Такты	Тактов на байт
ld (hl), a + inc hl	1	13	13,0
ld (hl), a + inc l	1	11	11,0
ldi	1	16	16,0
ldir (за байт)	1	21	21,0
push hl	2	11	5,5

PUSH записывает два байта за 11 тактов — 5,5 такта на байт. Почти в 4 раза быстрее LDIR. Подвох: PUSH пишет туда, куда указывает SP, а SP — это обычно твой стек. Чтобы использовать PUSH как канал данных, нужно перехватить указатель стека.

Техника

Паттерн всегда один и тот же:

1. Запретить прерывания (DI). Если прерывание сработает, пока SP указывает на экран, процессор протолкнёт адрес возврата в пиксельные данные. Наступит хаос.
2. Сохранить SP. Использовать самомодифицирующийся код, чтобы спрятать его.
3. Установить SP в *конец* целевой области. Стек растёт вниз — PUSH декрементирует SP перед записью. Поэтому если хочешь заполнить с \$4000 до \$57FF, установи SP в \$5800.
4. Загрузить данные в регистровые пары и PUSHить многократно.
5. Восстановить SP и разрешить прерывания (EI).



Почему PUSH выигрывает: LD (HL),A + INC HL записывает 1 байт за 13T (13,0 Т/байт). PUSH HL записывает 2 байта за 11T (**5,5 Т/байт**) — почти в 2,4 раза быстрее на байт. Цена: прерывания должны быть

отключены, пока SP перехвачен.

Вот суть примера `push_fill.a80` из каталога `examples/` этой главы:

```
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 | 16 PUSHes = 32 bytes
    push hl                 ; 11 T | = 176 T-states
    push hl                 ; 11 T |
    djnz .loop              ; 13 T (8 on last)

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

Внутреннее тело из 16 PUSH записывает 32 байта за 176 тактов. Итого для всей пиксельной области в 6 144 байта: примерно 36 000 тактов. Сравни с LDIR: $6\ 144 \times 21 - 5 = 129\ 019$ тактов. Метод PUSH примерно в 3,6 раза быстрее — разница между «укладывается в один кадр» и «вылезает в следующий».

POP как быстрое чтение

PUSH — самая быстрая запись, но POP — самое быстрое чтение. POP загружает 2 байта из (SP) в регистровую пару за 10 тактов — это 5,0 тактов на байт. Сравни альтернативы:

Метод	Прочитано байт	Такты	Тактов на байт
ld a, (hl) + inc hl	1	13	13,0
ld a, (hl) + inc l	1	11	11,0
ldi (как чтение+запись)	1	16	16,0
pop hl	2	10	5,0



Рис. 9: Заливка экрана через PUSH — вся пиксельная область заполнена за один кадр с помощью стекового трюка

Паттерн: заранее строишь в памяти таблицу 16-битных значений, направляешь SP на начало таблицы и выполняешь POP в регистровые пары. Каждый POP продвигает SP на 2, автоматически проходя по таблице. Это дополнение к PUSH-трюку записи на стороне чтения.

Скombинириуй POP и PUSH — и получишь быстрый конвейер «память-в-память»: POP значения из исходной таблицы (10T), при необходимости обработай регистровую пару, затем PUSH в место назначения (11T). Итого: 21 такт на 2 байта — та же пропускная способность, что у LDIR, но с регистровой парой, доступной для обработки между чтением и записью. Ты можешь маскировать биты, прибавлять смещения, менять байты местами или применять любое преобразование «регистр-в-регистр» без дополнительных расходов на доступ к памяти. Этот конвейер POP-обработка-PUSH — основа многих процедур скомпилированных спрайтов.

Где используются PUSH-трюки

- **Очистка экрана.** Самое распространённое применение. Каждому демо нужно очищать экран между эффектами.
- **Скомпилированные спрайты.** Спрайт компилируется в последовательность инструкций PUSH с предзагруженными регистровыми парами. Максимально быстрый вывод спрайтов на Z80.
- **Быстрый вывод данных.** Когда нужно перебросить блок данных в непрерывный диапазон адресов: заливки атрибутов, копирования буферов, построение списков вывода.

Цена: прерывания выключены. Если музыкальный плеер работает через прерывание IM2, он пропустит удар во время длинной последовательности PUSH. Демо-кодеры планируют с учётом этого — размещают PUSH-заливки во время бордюра или разбивают их на несколько кадров.

LDI-цепочки

LDI vs LDIR

LDI копирует один байт из (HL) в (DE), инкрементирует оба и декрементирует BC. LDIR делает то же самое, но повторяет до BC = 0. Разница — тайминг:

Инструкция	Такты	Примечания
LDI	16	Копирует 1 байт, всегда 16 Т
LDIR (за байт)	21	Копирует 1 байт, возвращается назад. Последний байт: 16 Т

LDIR стоит на 5 тактов больше за байт из-за внутренней проверки зациклиивания. Эти 5 тактов быстро накапливаются.

Для 256 байт: - LDIR: $255 \times 21 + 16 = 5\ 371$ такт - 256 x LDI: $256 \times 16 = 4\ 096$ тактов - Экономия: 1 275 тактов (24%)

Цепочка из отдельных инструкций LDI — это просто 256 повторений двухбайтового опкода \$ED \$A0. Это 512 байт кода ради 24% экономии — тот же размен RAM на скорость, что и при развертке циклов.

Когда LDI-цепочки блистают

Оптимальный случай — копирование блоков известного размера. Цепочка из 32 LDI экономит 160 тактов по сравнению с LDIR для строки спрайта. На 24 строках это 3 840 тактов за кадр.

Но настоящая мощь проявляется при комбинации LDI-цепочек с *арифметикой точек входа*. Если у тебя цепочка из 256 LDI и нужно скопировать только 100 байт, прыгай в цепочку на позицию 156. Никакого счётчика цикла, никакой настройки. Эта техника использована в хаотическом зумере Introspec'a в Eager (2015):

```
; 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
```

Это копирование переменной длины с нулевыми накладными расходами за байт — техника, которую принципиально невозможно реализовать с LDIR. Это одна из причин, почему LDI — лучший друг в демосценовом коде.



Рис. 10: LDI-цепочка vs LDIR — красные полосы показывают тайминг LDI-цепочки, синие — LDIR; более тонкие красные полосы доказывают, что LDI быстрее

Битовые трюки: SBC A,A и компания

SBC A,A как условная маска

После любой инструкции, устанавливающей флаг переноса, SBC A,A преобразует этот флаг в полный байт: \$FF, если перенос был установлен, 00, . : 4. — — ` JRC,.set`/`LDA,0`/`JR.done`/.set : LDA,FF/.done:` — которая стоит 17-22 такта в зависимости от пути, плюс нарушение конвейера из-за условного перехода.

Каноническое применение — *раскрытие бита в байт*. Имея байт, где каждый бит представляет пиксель (пиксельный формат Spectrum), можно раскрыть каждый бит в полный байт атрибута:

```
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
```

Три инструкции, 26 тактов, без ветвлений. Чтобы выбирать между двумя произвольными значениями, а не нулём и маской, используй паттерн SBC A,A : AND mask : XOR base. AND выбирает, какие биты различаются между двумя

значениями, а XOR переключает их на нужную базу. Этот паттерн заменяет каждую проверку «если бит установлен, то значение А, иначе значение В» во внутренних циклах.

ADD A,A vs SLA A

Обе инструкции сдвигают А влево на один бит. Но ADD A,A — это 4 такта и 1 байт, тогда как SLA A — 8 тактов и 2 байта. Нет ситуации, в которой SLA A предпочтительнее — ADD A,A строго быстрее и компактнее. Аналогично, ADD HL,HL сдвигает HL влево за 11 тактов (1 байт), заменяя двухинструкционную последовательность SLA L : RL H за 16 тактов (4 байта). Для 16-битного сдвига влево во внутреннем цикле, выполняемом 192 раза за кадр, одна эта замена экономит 960 тактов — более четырёх строк развёртки бордюрного времени.

Это не трюки. Это словарный запас. Точно так же, как свободно владеющий языком не останавливается, чтобы проспрашивать обычный глагол, Z80-программист тянется к ADD A,A и SBC A,A без осознанных усилий. Если ты ловишь себя на том, что пишешь SLA A или условный переход для выбора между двумя значениями, остановись и возьми более короткую форму. Такты складываются.

Генерация кода

Генерация кода: написание программы, которая рисует

Всё вышеописанное — фиксированные оптимизации: код работает одинаково в каждом кадре. Генерация кода идёт дальше: твоя программа пишет программу, которая рисует экран. Есть два варианта: офлайн (до ассемблирования) и рантайм (во время выполнения).

Оффлайн: генерация ассемблера из языка высокого уровня

Introspec использовал Processing (среду для творческого кодирования на Java) для генерации Z80-ассемблера хаотического зумера в Eager (2015). Хаотический зумер меняет масштаб каждый кадр — разные исходные пиксели отображаются на разные экранные позиции. Вместо вычисления этих отображений в реальном времени скрипт Processing предвычислял каждое отображение и выдавал исходные файлы .a80, содержащие оптимизированные LDI-цепочки и инструкции LD.

Рабочий процесс: скрипт Processing вычисляет для каждого кадра, какой байт источника отображается на какой байт экрана. Он выдаёт исходный код Z80 — последовательности ld hl, source_addr и ldi — которые ассемблер (sjasmplus) собирает вместе с рукописным кодом движка. При выполнении движок просто вызывает предсгенерированный код для текущего кадра.

Это не «жульничество». Это фундаментальное понимание того, что разделение труда между временем компиляции и временем выполнения может полностью убрать ветвления, поиск и арифметику из внутреннего цикла. Скрипт Processing делает тяжёлую математику один раз, медленно, на современной машине. Z80 делает простую часть — копирование байтов — с максимально возможной скоростью.

Рантайм: программа пишет машинный код во время выполнения

Иногда параметры меняются каждый кадр, и офлайн-генерации недостаточно. Процедура отображения сферы в Illusion от X-Trade (ENLiGHT'96) генерирует машинный код в буфер ОЗУ в реальном времени. Геометрия сферы меняется при вращении — разным пикселям нужны разные расстояния пропуска. Перед каждым кадром движок эмитирует байты опкодов в буфер, а затем исполняет их:

```
; 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
```

Сгенерированный код — линейная последовательность без ветвлений, без поиска, без накладных расходов цикла, но это *разный код каждый кадр*. Вместо «if pixel_skip == 3 then...» по 12 тактов на ветвление ты эмитируешь точно нужные инструкции и исполняешь их без ветвлений.

Стоимость генерации

Генерация кода во время выполнения не бесплатна. Посмотри на цикл генератора выше: на каждую эмитированную инструкцию нужно загрузить байт опкода, записать его, продвинуть указатель и, возможно, загрузить операнд — примерно 30-50 тактов на эмитированный байт, в зависимости от сложности. Примем ~40 тактов в среднем. Для сгенерированной процедуры из 100 байт инструкций это около 4 000 тактов накладных расходов на генерацию.

Точка безубыточности: генерация окупается, когда сгенерированный код выполняется более одного раза за кадр, или когда он заменяет логику ветвления,

которая стоит дороже самой генерации. В сферическом маппере Illusion каждый сгенерированный проход рендеринга выполняется один раз за кадр — но он заменяет попиксельные условные переходы, которые стоили бы куда дороже. Alone Coder документировал аналогичный компромисс в своём движке вращения: генерация последовательности инструкций INC H/INC L для пошагового прохода по координатам стоит примерно 5 000 тактов на эмиссию, но устраняет координатную арифметику, которая обошлась бы примерно в 146 000 тактов при онлайн-вычислении. Накладные расходы генерации — менее 4% от стоимости, которую она заменяет.

Эмпирическое правило: если ты обнаруживаешь, что пишешь цикл с ветвлением, выбирающими между разными последовательностями инструкций на основе попиксельных или построчных данных, этот цикл — кандидат на генерацию кода. Эмитируй правильные инструкции один раз, выполни их без ветвлений и перегенерируй только при изменении параметров.

Когда генерировать код: Если одни и те же операции выполняются каждый кадр и меняются только данные, самомодифицирующегося кода (подмены операндов) достаточно. Если меняется *структура* — разное число итераций, разные последовательности инструкций — генерируй код. Если вариации можно предвычислить на современной машине, предпочтай онлайн-генерацию: она отлаживаема, верифицируема и не стоит ничего в рантайме. Рантайм-генерация окупается, когда сгенерированный код выполняется гораздо чаще, чем стоит его генерация.

RET-цепочки

Превращение стека в таблицу диспетчеризации

В 2025 году DenisGrachev опубликовал на Нуре технику, разработанную для его игры Dice Legends. Задача: отрисовка тайлового игрового поля требует вывода десятков тайлов за кадр. Наивный подход использует CALL:

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

Каждый CALL стоит 17 тактов. Для игрового поля 30 x 18 (540 тайлов) это 9 180 тактов только на диспетчеризацию.

Идея DenisGrachev'a: установить SP на *список рендеринга* — таблицу адресов — и завершать каждую процедуру отрисовки тайла инструкцией RET. RET извлекает 2 байта из (SP) в PC. Если SP указывает на твой список рендеринга, RET не возвращается к вызывающему — он переходит к следующей процедуре в списке.

```
; 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

```

Теперь каждая диспетчеризация стоит 10 тактов (RET) вместо 17 (CALL). Для 540 тайлов: экономия 3 780 тактов. Но настоящий выигрыш — бесплатная диспетчеризация: каждая запись может указывать на разную процедуру (широкий тайл, пустой тайл, анимированный тайл). Никакой таблицы переходов, никакого косвенного вызова. Список рендеринга *и есть* программа.

Три стратегии для списка рендеринга

DenisGrachev исследовал три подхода к построению списка рендеринга:

- Карта как список рендеринга.** Тайловая карта сама является списком рендеринга: каждая ячейка содержит адрес процедуры отрисовки для данного типа тайла. Просто, но негибко — смена тайла означает перезапись 2 байт в карте.
- Сегменты на основе адресов.** Экран делится на сегменты. Список рендеринга каждого сегмента — блок адресов, скопированный из мастер-таблицы. Смена тайлов — копирование нового блока адресов.
- Байтовый подход с 256-байтовыми таблицами поиска.** Каждый тип тайла — один байт (индекс тайла). 256-байтовая таблица поиска отображает индексы тайлов в адреса процедур. Список рендеринга строится проходом по байтам тайловской карты с поиском каждого адреса. Этот подход DenisGrachev выбрал для Dice Legends.

Используя байтовый подход, он расширил игровое поле с 26 x 15 тайлов (предел предыдущего движка) до 30 x 18 тайлов, сохраняя целевую частоту кадров. Экономия от устранения накладных расходов CALL в сочетании с бесплатной диспетчеризацией освободила достаточно тактов для рендеринга на 40% больше тайлов.

Компромиссы

Как и все стековые трюки, прерывания должны быть запрещены, пока SP перехвачен. Каждая процедура тайла должна быть самодостаточной — заканчиваться RET и не использовать CALL, поскольку настоящий стек недоступен. На практике процедуры тайлов достаточно коротки, чтобы это не было ограничением.

Врезка: «Код мёртв» (Introspec, 2015)

В январе 2015 года Introspec опубликовал на Hure короткое, провокационное эссе «Код мёртв» (Kod myortv). Аргумент проводит параллель с «Смертью автора» Ролана Барта: подобно тому как Барт утверждал, что смысл текста принадлежит читателю, а не автору, Introspec утверждает, что демо-код по-настоящему живёт только тогда, когда кто-то его читает — в отладчике, в листинге дизассемблера, в исходниках, расшаренных на форуме.

Неудобная правда: современные демо потребляются как визуальный контент. Люди смотрят их на YouTube. Голосуют на Pouet по видеозаписям. Никто не видит внутренние циклы. Блестящая оптимизация, экономящая 3 такта на пиксель, невидима для 99% аудитории. «Писать код чисто ради него самого», — написал Introspec, — «потеряло актуальность».

И всё же.

Ты читаешь эту книгу. Мы открываем отладчик. Мы считаем такты. Мы заглядываем внутрь. Техники в этой главе — не музейные экспонаты. Это живые инструменты, и то, что большинство людей их никогда не увидит, не умаляет их мастерства.

Эссе Introspec'а — это вызов, а не капитуляция. После него он опубликовал одни из самых детальных технических анализов, которые ZX-сцена когда-либо видела — включая разбор Illusion и бенчмарки компрессии, упоминаемые на протяжении всей этой книги. Код может быть мёртв для зрителя YouTube. Но для читателя с дизассемблером и пытливым умом он очень даже жив.

Собираем всё вместе

Техники в этой главе не независимы. На практике они комбинируются:

- **Очистка экрана** сочетает *развёрнутые циклы* с *PUSH-трюками*: частично развёрнутый цикл из 16 PUSH за итерацию, с SP, перехваченным через *самомодифицирующийся код*.
- **Скомпилированные спрайты** сочетают *генерацию кода* (каждый спрайт компилируется в исполняемый код), *чтение через POP и вывод через PUSH* (самый быстрый способ перемещать пиксельные данные через регистры), *битовые трюки* (SBC A,A для раскрытия маски) и *самомодификацию* (подмена экранных адресов каждый кадр).

- **Тайловые движки** сочетают RET-цепочки для диспетчеризации с LDI-цепочками внутри каждой процедуры тайла для быстрого копирования данных.
- **Хаотические зумеры** сочетают онлайн-генерацию кода (скрипты Processing, эмитирующие ассемблер) с LDI-цепочками (сгенерированный код — в основном последовательности LDI) и самомодификацией (подмена адресов источника каждый кадр).
- **Атрибутные эффекты** сочетают чтение через POP из предвычисленных таблиц с битовыми трюками (SBC A,A для раскрытия битовых масок в цветовые значения) и записью через PUSH для быстрого вывода атрибутов.

Общая нить: каждая техника убирает что-то из внутреннего цикла. Развёртка убирает счётчик цикла. Самомодификация убирает ветвления. PUSH убирает побайтовые накладные расходы на запись. POP убирает побайтовые накладные расходы на чтение. LDI-цепочки убирают штраф повтора LDIR. Битовые трюки убирают условные ветвления. Генерация кода убирает само различие между кодом и данными. RET-цепочки убирают накладные расходы CALL.

Z80 работает на 3,5 МГц. У тебя 71 680 тактов на кадр. Каждый такт, который ты сэкономишь во внутреннем цикле — это такт, который можно потратить на больше пикселей, больше цветов, больше движения. Инструментарий этой главы — вот как туда добраться.

В последующих главах ты увидишь каждую из этих техник в работе в реальных демо — текстурированная сфера Illusion, атрибутный туннель Eager, мультиколорный движок Old Tower. Цель этой главы — дать тебе словарь. Теперь посмотрим, что мастера с ним построили.

Попробуй сам

1. **Измерь разницу.** Возьми тестовую обвязку из главы 1 и измерь три версии заполнения 256 байт: (a) цикл ld (hl), a : inc hl : djnz, (b) полностью развёрнутая ld (hl), a : inc hl x 256, и (c) PUSH-заливка из examples/push_fill.a80. Сравни ширину бордюрных полос. Полоса PUSH-версии должна быть заметно короче.
2. **Построй самомодифицирующуюся очистку.** Напиши процедуру очистки экрана, принимающую паттерн заливки как параметр и подставляющую его в PUSH-цикл заливки через самомодифицирующийся код. Вызови её дважды с разными паттернами и наблюдай, как экран чередуется.
3. **Замерь LDI-цепочку.** Напиши 32-байтовое копирование с помощью LDIR и другое с помощью 32 x LDI. Измерь оба бордюрным методом. LDI-цепочка должна сэкономить 160 тактов — это заметно, если запускать копирование в тесном цикле.
4. **Поэкспериментируй с точками входа.** Построй LDI-цепочку из 128 записей и маленькую процедуру, вычисляющую точку входа по значению в регистре A (0-128). Прыгай в цепочку в разных точках. Это упрощённая версия копирования переменной длины, используемого в настоящих хаотических зумерах.

5. **Копировщик переменной длины с вычисляемой точкой входа.** Построй LDI-цепочку из 256 элементов и фронтенд, принимающий количество байт в регистре B (1-256). Вычисли точку входа: каждый LDI занимает 2 байта, значит смещение равно $(256 - B) \times 2$ от начала цепочки. Прибавь его к базовому адресу цепочки, затем JP (HL) в неё. Оберни всё это в тестовую обвязку с цветом бордюра и сравни ширину полосы с LDIR для того же количества байт. Для малых значений (менее 16) разница невелика. Для значений выше 64 LDI-цепочка заметно вырывается вперёд.
6. **Распаковщик бита в атрибут.** Напиши подпрограмму, которая читает байт из (HL), вращает каждый бит наружу через RLC (HL) и использует SBC A,A : AND \$47 для раскрытия каждого бита в байт атрибута (ярко-белый или чёрный). Сохрани 8 результирующих байт атрибутов в буфер назначения через (DE) / INC DE. Это зародыш записи атрибутов скомпилированного спрайта — в следующих главах ты увидишь, как этот паттерн генерирует целевые спрайтовые процедуры.

Источники: DenisGrachev «Tiles and RET» (Hype, 2025); Introspec «Making of Eager» (Hype, 2015); Introspec «Technical Analysis of Illusion» (Hype, 2017); Introspec «Код мёртв» (Hype, 2015)

Глава 4: Математика, которая реально нужна

«Прочтайте учебник математики — производные, интегралы.
Они вам понадобятся.» – Dark, Spectrum Expert #01 (1997)

В 1997 году подросток в Санкт-Петербурге сел писать журнальную статью об умножении. Не о том, которому учат в школе, — а о том, которое заставляет каркасный куб вращаться на ZX Spectrum со скоростью 50 кадров в секунду. Его звали Dark, он кодил в группе X-Trade, и его демо *Illusion* уже заняло первое место на ENLiGHT'96. Теперь он писал *Spectrum Expert*, электронный журнал, распространяемый на дискетах, и собирался подробно объяснить, как работают его алгоритмы.

Всё, что следует далее, взято непосредственно из статьи Dark'a «Алгоритмы программирования» в *Spectrum Expert #01*. Это подпрограммы, которые приводили в движение *Illusion* — то самое умножение, что вращало вершины, та самая таблица синусов, что управляла ротозумером, тот самый рисовальщик линий, что рендерил каркасные фигуры на полной частоте кадров. Когда Introspec двадцать лет спустя дизассемблировал *Illusion* в блоге Нура, он обнаружил именно эти алгоритмы в работе внутри бинарника.

Умножение на Z80

У Z80 нет инструкции умножения. Каждый раз, когда тебе нужно A умножить на B — для матриц вращения, перспективной проекции, текстурного маппинга — приходится синтезировать умножение из сдвигов и сложений. Dark представляет два метода и с характерной честностью рассуждает о компромиссе между ними.

Метод 1: Сдвиг-и-сложение от LSB

Классический подход. Проходим по битам множителя от младшего к старшему. Для каждого установленного бита прибавляем множимое к аккумулятору. После каждого бита сдвигаем аккумулятор вправо. После восьми итераций аккумулятор содержит полное произведение.

Вот беззнаковое умножение 8x8 от Dark'a. Вход: В умножить на С. Результат в А (старший байт) и С (младший байт):

```
; 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

```

Изучи это внимательно. Инструкция RRA сдвигает А вправо, но также выталкивает младший бит А в флаг переноса. На следующей итерации RR С вдвигает этот перенос в старший бит С. Таким образом, младшие биты произведения постепенно собираются в С, а старшие накапливаются в А. После восьми итераций полный 16-битный результат лежит в А:С.

Стоимость — от 196 до 204 тактов (T-state) в зависимости от того, сколько бит множителя установлено: каждый установленный бит добавляет один ADD A,B (4 такта). Пример в chapters/ch04-maths/examples/multiply8.a80 показывает вариант с результатом в HL.

Для 16x16 с 32-битным результатом подпрограмма Dark'a MULU224 работает за 730-826 тактов. На практике демосценовые 3D-движки избегают полного 16x16 умножения, храня координаты в формате с фиксированной точкой 8.8 и используя умножение 8x8 где возможно.

Метод 2: Поиск по таблице квадратов

Второй метод Dark'a обменивает память на скорость, используя алгебраическое тождество, которое рано или поздно открывает каждый демосценер:

Заранее вычисляем таблицу значений $n^{2/4}$, и умножение становится двумя поисками по таблице и вычитанием — примерно 61 такт, более чем втрое быстрее сдвига-и-сложения.

Нужна 512-байтная таблица ($n^{2/4}$) для $n = 0..511$, выровненная по странице для индексации одним регистром. Таблица должна быть 512 байт, потому что $(A+B)$ может достигать 510.

```

; MULU_FAST -- Square table multiply
; Input:  B, C = unsigned 8-bit factors

```

Shift-and-Add Multiply: $13 \times 11 = 143$

Multiplier = 13 = 00001101 Multiplicand = 11 = 00001011

Step	Bit	Multiplier bits	Action	Acc	Acc (binary)
0	b0 = 1	0 0 0 0 1 1 0 1	ADD 11	11	00001011
1	b1 = 0	0 0 0 0 1 1 0 1	skip	11	00001011
2	b2 = 1	0 0 0 0 1 1 0 1	ADD 44	55	00110111
3	b3 = 1	0 0 0 0 1 1 0 1	ADD 88	143	10001111
4	b4 = 0	0 0 0 0 1 1 0 1	skip	143	10001111
5	b5 = 0	0 0 0 0 1 1 0 1	skip	143	10001111
6	b6 = 0	0 0 0 0 1 1 0 1	skip	143	10001111
7	b7 = 0	0 0 0 0 1 1 0 1	skip	143	10001111

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

Bit set: ADD multiplicand

Bit clear: skip (no addition)

Рис. 11: Shift-and-add 8x8 multiply walkthrough

```

; 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

```

Компромисс? Dark с характерной честностью говорит: «**Выбирай: скорость или точность.**» Таблица хранит целочисленные значения $n^2/4$, поэтому ошибка округления составляет до 0.25 на каждый поиск. Для больших значений это пренебрежимо мало. Для малых приращений координат при 3D-вращении ошибка вызывает заметное дрожание вершин. С методом сдвига-и-сложения вращение идеально плавное.

Для текстурного маппинга, плазмы, скроллеров — используй быстрое умножение. Для каркасного 3D, где глаз отслеживает отдельные вершины — оставайся на сдвиге-и-сложении. Dark знал это, потому что пробовал оба варианта в *Illusion*.

Генерация таблицы квадратов — одноразовая затрата при инициализации. Dark предлагает метод производных: поскольку $d(x^2)/dx = 2x$, можно строить таблицу инкрементально, прибавляя линейно растущую дельту на каждом шаге. На практике большинство кодеров вычисляют таблицу в BASIC-загрузчике или подпрограмме инициализации и идут дальше.

Знаковое умножение

Глава, которая учит беззнаковому умножению, а затем использует его для вращения 3D-координат, имеет пробел: матрицы вращения оперируют знаковыми значениями. X может быть -40 или +40, значения синуса лежат в диапазоне от -128 до +127. Каждый `call mul_signed` в главе 5 зависит от подпрограммы, которую ты сейчас увидишь. Как выразился Ped7g в своём ревью: «глава, которая учит 3D-вращению, не показывая знаковое умножение, — это как кулинарная книга, которая перечисляет ингредиенты, но забывает про духовку».

Дополнительный код на практике

Z80 представляет знаковые целые числа в дополнительном коде. Правила просты:

- Бит 7 — знаковый бит: 0 = положительное, 1 = отрицательное
- Положительные значения совпадают с беззнаковыми: \$00 = 0, \$01 = 1, ..., \$7F = 127
- Отрицательные значения отсчитываются от \$FF: \$FF = -1, \$FE = -2, ..., \$80 = -128
- NEG вычисляет абсолютное значение отрицательного числа (отрицание A: A = 0 - A). Цена: 8T

Критически важный момент для арифметики: **ADD и SUB не зависят от знаковости**. Сложение \$FF (-1) с \$03 (+3) даёт \$02 (+2) — корректно и в знаковой, и в беззнаковой интерпретации. Аппаратное сложение идентично. Только умножение требует явной обработки знака, потому что цикл сдвига и сложения трактует биты множителя как беззнаковые позиционные значения.

Расширение знака: идиома `rla / sbc a,a`

Когда ты умножаешь 8-битное знаковое значение на другое 8-битное знаковое значение, тебе нужно знать знаки. Самый дешёвый способ извлечь знаковый бит на Z80:

```
; Cost: 8T, 2 bytes. Branchless.
    rla          ; 4T rotate sign bit into carry
    sbc a, a     ; 4T A = 0 if carry clear, $FF if set
```

После `sbc a,a` в A находится \$00 для положительных входов или \$FF для отрицательных. Это стандартный байт расширения знака, используемый по всей Z80-демосцене.

`mul_signed` — знаковое умножение 8x8

Алгоритм: XOR двух входов для определения знака результата, взятие абсолютных значений, беззнаковое умножение, отрицание результата, если знак был отрицательным. Это подпрограмма, которую глава 5 вызывает шесть раз на вращение вершины и дважды на отсечение задней грани.

```
; mul_signed - 8x8 signed multiply
; Input:  B = signed multiplicand, C = signed multiplier
; Output: HL = signed 16-bit result
; Cost: ~240-260 T-states (Pentagon)
```

```

;

; Algorithm: determine sign, abs both, unsigned multiply, negate if needed.

mul_signed:
    ld   a, b
    xor  c           ; 4T bit 7 = result sign (1 = negative)
    push af          ; 11T save sign flag

    ; Absolute value of B
    ld   a, b
    or   a
    jp   p, .b_pos   ; 10T skip if positive
    neg              ; 8T A = |B|
.b_pos:
    ld   b, a

    ; Absolute value of C
    ld   a, c
    or   a
    jp   p, .c_pos
    neg
.c_pos:
    ld   c, a

    ; Unsigned 8x8 multiply: B * C -> A:C (high:low)
    ld   a, 0
    ld   d, 8
.mul_loop:
    rr   c
    jr   nc, .noadd
    add  a, b
.noadd:
    rra
    dec  d
    jr   nz, .mul_loop

    ; A:C = unsigned product. Move to HL.
    ld   h, a
    ld   l, c

    ; Negate result if sign was negative
    pop  af          ; 10T recover sign
    or   a
    jp   p, .done     ; 10T skip if positive
    ; Negate HL: HL = 0 - HL
    xor  a
    sub  l
    ld   l, a
    sbc  a, a
    sub  h
    ld   h, a
.done:

```

```
ret
```

Ядро — тот же цикл сдвига и сложения из `mulu112`, обёрнутый определением знака и условным отрицанием. Накладные расходы — ~40-60 тактов сверх беззнакового умножения, в зависимости от того, сколько операндов требуют отрицания.

`mul_signed_c` — тонкая обёртка для отсечения задних граней

Отсечение задних граней в главе 5 передаёт первый operand в A, а не в B. Тонкая обёртка позволяет не перестраивать вызывающий код:

```
; mul_signed_c – signed multiply with A,C inputs
; Input: A = signed multiplicand, C = signed multiplier
; Output: HL = signed 16-bit result
; Cost: ~250-270 T-states (Pentagon)
```

```
mul_signed_c:
    ld   b, a           ; 4T
    jr   mul_signed     ; 12T fall through to mul_signed
```

Сравнение стоимости

Подпрограмма	Вход	Результат	Такты	Примечания
<code>mulu112</code> (беззнаковая)	B, C	A:C (16-бит)	196-204	Глава 4, умножение сдвигом и сложением
<code>mulu_fast</code> (таблица квадратов)	B, C	HL (16-бит)	~61	Нужна таблица 512 байт; ошибка округления
<code>mul_signed</code>	B, C (знаковые)	HL (знаковый 16-бит)	~240-260	Обработка знака добавляет ~40-60T
<code>mul_signed_c</code>	A, C (знаковые)	HL (знаковый 16-бит)	~250-270	Обёртка для отсечения задних граней

Знаковое умножение примерно на 25% дороже беззнакового. Для каркасного куба с 8 вершинами и 6 умножениями на поворот оси (12 всего на полное 3D-вращение) стоимость на вершину составляет ~3 120 тактов — всё ещё комфортно укладывается в бюджет кадра.

Матрицы вращения в главе 5 вызывают `mul_signed` шесть раз на вершину для Z-оси и перспективы, а `mul_signed_c` — дважды на грань для отсечения задних граней. Теперь ты точно знаешь, что делают эти вызовы.

Благодарность: Пробел в знаковой арифметике выявил Ped7g (Peter Helcmanovsky) в ходе ревью книги.

Деление на Z80

Деление на Z80 ещё мучительнее умножения. Нет инструкции деления, и алгоритм по своей природе последователен — каждый бит частного зависит от предыдущего вычитания. Dark снова представляет два метода: точный и быстрый.

Метод 1: Сдвиг-и-вычитание (восстановливающее деление)

Двоичное деление столбиком. Начинаем с обнулённого аккумулятора. Делимое сдвигается справа, по одному биту за итерацию. Пробуем вычесть делитель; если получается — устанавливаем бит частного. Если нет — восстанавливаем аккумулятор, отсюда и название «восстановливающее деление».

```
; 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
```

INC B для установки бита частного — изящный трюк: В только что был сдвинут влево инструкцией SLA B, поэтому бит 0 гарантированно нулевой. INC B устанавливает его, не затрагивая остальные биты — дешевле, чем OR или SET.

16-битная версия (DIVU222) стоит от 938 до 1034 тактов. Тысяча тактов на одно деление. При бюджете кадра ~70 000 тактов можно позволить себе примерно 70 делений на кадр — не делая больше ничего. Вот почему демосценовые 3D-движки идут на крайние меры, чтобы избежать деления.

Метод 2: Логарифмическое деление

Более быстрая альтернатива Dark'a использует таблицы логарифмов:

$$A / B = \text{AntiLog}(\text{Log}(A) - \text{Log}(B))$$

С двумя 256-байтными таблицами подстановки — Log и AntiLog — деление превращается в два поиска по таблице, вычитание и третий поиск. Стоимость падает примерно до 50-70 тактов. Для перспективного деления (деление на Z для проецирования 3D-точек на экран) — это революция.

Генерация таблицы логарифмов — тут становится интересно. Dark предлагает строить её через производные — тот же инкрементальный метод, что и для таблицы квадратов. Производная $\log_2(x)$ равна $1/(x * \ln(2))$, поэтому ты накапливаешь дробные приращения шаг за шагом, начиная с $\log_2(1) = 0$ и двигаясь вверх. Константу $1/\ln(2) = 1.4427$ нужно масштабировать, чтобы уместить в 8-битный диапазон таблицы.

И тут проявляется честность Dark'a. Выведя формулу генерации, он пытается вычислить поправочный коэффициент для масштабирования таблицы и получает 0.4606. Затем пишет — в публикуемой журнальной статье — «*Что-то тут не так, поэтому рекомендуется написать аналогичную самостоительно.*»

Семнадцатилетний парень в 1997 году, публикуясь в дисковом журнале, который читают его коллеги по всей российской Spectrum-сцене, открыто говорит: я заставил это работать, но в моём выводе есть дыра — разберитесь с чистой версией сами. Такая честность редка в техническом тексте любого уровня, и это одна из вещей, которые делают Spectrum Expert таким замечательным документом.

На практике таблицы логарифмов работают. Ошибки округления при сжатии непрерывной функции в 256 байт приемлемы для перспективной проекции. 3D-движок Dark'a в *Illusion* использует именно этот метод.

Синус и косинус

Вращение, скроллинг, плазма — каждому эффекту с кривизной нужна тригонометрия. На Z80 ты заранее вычисляешь таблицу подстановки. Подход Dark'a красиво прагматичен: парабола достаточно близка к синусоиде для демосценовых целей.

Параболическая аппроксимация

Половина периода косинуса, от 0 до π , изгибаются от +1 вниз до -1. Парабола $y = 1 - 2*(x/\pi)^2$ следует почти тем же путём. Максимальная ошибка — около 5.6% — ужасно для инженерных расчётов, незаметно в демо при разрешении 256x192.

Dark генерирует 256-байтную знаковую таблицу косинуса (-128..+127), индексируемую углом: 0 = 0 градусов, 64 = 90 градусов, 128 = 180 градусов, 256 оборачивается в 0. Период, равный степени двойки, означает, что индекс угла оборачивается естественно при 8-битном переполнении, а косинус становится синусом прибавлением 64.

```

; 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.

```

Ключевая идея: не нужно вычислять x^2 для каждой записи. Поскольку $(x+1)^2 - x^2 = 2x + 1$, ты строишь параболу инкрементально — начинаясь с вершины, вычитаешь линейно растущую дельту. Никакого умножения, деления, никакой плавающей точки.

Получившаяся таблица — кусочно-параболическая аппроксимация. Наложи её на истинный синус, и разницу будет сложно увидеть. Для каркасного 3D или прыгающего скроллера — более чем достаточно.

Врезка: 9 заповедей Raider'a о таблицах синусов

В комментариях Нуре к анализу *Illusion Introspec*'ом ветеран-кодер Raider выложил список правил проектирования таблиц синусов, ставший неформально известным как «9 заповедей». Основные принципы:

- Используй размер таблицы, равный степени двойки (256 записей — каноничный размер).
- Выравнивай таблицу по границе страницы, чтобы Н хранил базу, а L был чистым углом — индексация бесплатна.
- Храни знаковые значения для прямого использования в координатной арифметике.
- Пусть угол оборачивается естественно через 8-битное переполнение — никаких проверок границ.
- Косинус — это просто синус со смещением на четверть периода: загрузи угол, прибавь 64, сделай поиск.
- Если нужна повышенная точность, используй 16-битную таблицу (512 байт), но это редко требуется.
- Генерируй таблицу при запуске, а не храни в бинарнике — экономит место, ничего не стоит.
- Для 3D-вращения предварительно умножь на масштабный коэффициент и храни масштабированные значения.

- Никогда не вычисляй тригонометрию в реальном времени. Если тебе кажется, что нужно — ты ошибаешься.

Эти заповеди отражают десятилетия коллективного опыта. Следуй им, и твои таблицы синусов будут быстрыми, компактными и корректными.

Рисование линий по Брезенхэму

Каждое ребро каркасного объекта — это линия от (x_1, y_1) до (x_2, y_2) , и рисовать её нужно быстро. Раздел Dark'a в Spectrum Expert #01, посвящённый рисованию линий, — самый объёмный в статье; он проходит через три последовательно ускоряющихся подхода.

Классический алгоритм и модификация Xopha

Алгоритм Брезенхэма шагает вдоль главной оси по одному пикселю за раз, поддерживая аккумулятор ошибки для шагов по второстепенной оси. На Spectrum «поставить пикセル» — дорогая операция: чересстрочная раскладка экрана означает, что вычисление адреса байта и битовой позиции стоит реальных тактов. Подпрограмма ПЗУ тратит более 1000 тактов на пикセル. Даже ручной оптимизированный цикл Брезенхэма стоит ~80 тактов на пикセル.

Dark упоминает улучшение Xopha: поддерживать указатель на экранную память (HL) и продвигать его инкрементально, а не пересчитывать с нуля. Движение вправо — ротация битовой маски; движение вниз — многоинструкционная корректировка DOWN_HL. Лучше, но коренная проблема остаётся.

Матричный метод Dark'a: сетки 8x8 пикселей

Затем Dark делает ключевое наблюдение: **«87.5% проверок тратятся впустую.»**

В цикле Брезенхэма на каждом пикселе ты спрашиваешь: нужно ли шагнуть вбок? Для почти горизонтальной линии ответ почти всегда — нет. В среднем семь из восьми проверок не дают бокового шага. Ты сжигаешь такты на условном переходе, который почти никогда не срабатывает.

Решение Dark'a: предвычислить паттерн пикселей для каждого наклона линии в пределах сетки 8x8 пикселей и развернуть цикл рисования, чтобы выводить целые ячейки сетки за раз. Сегмент линии в пределах области 8x8 полностью определяется наклоном. Для каждого из восьми октантов перечисляются все возможные 8-пиксельные паттерны как прямые последовательности инструкций SET bit,(HL) с приращениями адреса между ними.

```
; 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)

```

Никаких условных переходов. Никакого аккумулятора ошибки. SET bit,(HL) занимает 15 тактов; восемь таких инструкций плюс пара INC H дают ~130 тактов на 8-пиксельный сегмент, или около 16 тактов на пиксель. С учётом поиска по таблице подстановки и перехода между ячейками Dark достигает примерно **48 тактов на пиксель** — почти вдвое дешевле классического Брезенхэма.

Цена — память: отдельная развёрнутая подпрограмма для каждого наклона в каждом октанте, всего около **3 КБ**. На 128K Spectrum — скромное вложение ради колоссального ускорения.

Завершение через ловушку

Вместо проверки счётчика цикла на каждом пикселе Dark размещает маркер-ловушку там, где линия заканчивается. Когда рисующий код натыкается на маркер, он выходит — полностью устранив накладные расходы DEC counter / JR NZ.

Полная система — выбор октанта, поиск сегмента, развёрнутое рисование, завершение через ловушку — один из самых впечатляющих фрагментов кода в Spectrum Expert #01. Когда Introspec дизассемблировал *Illusion* в 2017 году, он обнаружил именно этот матричный метод за рисованием каркасных фигур на полной частоте кадров.

Арифметика с фиксированной точкой

Каждый алгоритм в этой главе подразумевает то, о чём мы ещё не сказали явно: числа с фиксированной точкой.

У Z80 нет блока плавающей точки. Каждый регистр хранит целое число. Но демо-эффектам нужны дробные значения — углы вращения, субпиксельные скорости, масштабные коэффициенты. Решение — фиксированная точка: выбираешь соглашение о том, где находится «десятичная точка» внутри целого числа, а затем выполняешь всю арифметику в целых числах, мысленно отслеживая масштаб.

Формат 8.8

Самый распространённый формат на Z80 — **8.8**: старший байт = целая часть, младший байт = дробная часть. Одна 16-битная регистровая пара хранит одно число с фиксированной точкой:

`L = fractional part (0..255, representing 0/256 to 255/256)`

`HL = $0180` представляет 1.5 ($H=1$, $L=128$, и $128/256 = 0.5$). `HL = $FF80` знаковое — это -0.5 ($H=$FF = -1$ в дополнительном коде, $L=$80$ добавляет 0.5).

Красота в том, что **сложение и вычитание бесплатны** — обычные 16-битные операции:

```
; 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.
```

Процессору всё равно, что ты трактуешь эти числа как фиксированную точку. Двоичное сложение одинаково, представляют ли биты целые числа или значения 8.8.

Умножение с фиксированной точкой

Умножение двух чисел 8.8 даёт результат 16.16 — 32 бита. Тебе нужно обратно 8.8, поэтому берёшь биты 8..23 произведения (фактически сдвигая вправо на 8). На практике, с малыми целыми частями (координаты, коэффициенты вращения между -1 и +1), можно разложить умножение на частичные произведения:

```
; 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

```

Для вращения на базе таблиц синусов, где значения синуса — 8-битные знаковые (-128..+127, представляющие -1.0..+0.996), умножение 8-битной координаты на значение синуса через `mulu112` даёт 16-битный результат уже в формате 8.8 — старший байт — повёрнутая целая координата, младший — дробная часть.

Почему фиксированная точка важна

Формат 8.8 — оптимальный выбор для Z80: умещается в регистровую пару, сложение/вычитание бесплатны, умножение стоит ~200 тактов, и точности хватает для эффектов экранного разрешения. Существуют и другие форматы — 4.12 для большей дробной точности, 12.4 для большего целочисленного диапазона — но 8.8 покрывает подавляющее большинство случаев. Главы по разработке игр далее в этой книге используют исключительно 8.8.

Теория и практика

Эти алгоритмы — не изолированные приёмы. Они образуют систему. Умножение питает матрицу вращения. Вращение выдаёт координаты, нуждающиеся в перспективном делении. Деление использует таблицы логарифмов. Спроектированные вершины соединяются линиями, нарисованными матричным методом. Всё работает на арифметике с фиксированной точкой, со значениями синуса из параболической таблицы.

Dark проектировал их как компоненты единого движка — движка, приводившего в действие *Illusion*. Каркасный куб, вращающийся на полной частоте кадров, задействует каждую подпрограмму из этой главы:

- Считываем угол вращения** из таблицы синусов (параболическая аппроксимация, ~20 тактов на поиск)
- Умножаем** координаты вершин на коэффициенты вращения (сдвиг-и-сложение для точности или таблица квадратов для скорости — ~200 или ~60 тактов на умножение, 12 умножений на вершину)
- Делим** на Z для перспективной проекции (таблицы логарифмов, ~60 тактов на деление)
- Рисуем линии** между спроектированными вершинами (матричный Брезенхэм, ~48 тактов на пиксель)

Для простого куба (8 вершин, 12 рёбер) суммарная стоимость на кадр составляет примерно:

- Вращение: 8 вершин \times 12 умножений \times 200 тактов = 19 200 тактов
- Проекция: 8 вершин \times 1 деление \times 60 тактов = 480 тактов
- Рисование линий: 12 рёбер \times ~40 пикселей \times 48 тактов = 23 040 тактов
- **Итого: ~42 720 тактов** — комфортно укладывается в бюджет кадра ~70 000 тактов

Переключись на быстрое умножение через таблицу квадратов, и вращение падает до 5 760 тактов. Вершины слегка дрожат, но зато появляется запас для более сложных объектов. Скорость или точность — в демо этот выбор делаешь для каждого эффекта, каждого кадра.

Что Dark сделал правильно

Оглядываясь на Spectrum Expert #01 с дистанции почти в тридцать лет, поражает не только качество алгоритмов, но и качество мышления. Dark представляет каждый из них, честно объясняет компромиссы, признаёт, когда в его выводе есть пробелы, и доверяет читателю заполнить их.

Он писал для Spectrum-кодеров в России конца 1990-х — сообщества, создававшего одни из самых впечатляющих 8-битных демо в мире на оборудовании, которое остальной мир уже забросил. Это строительные блоки, которые они использовали. Когда ты будешь писать свой первый 3D-движок для Spectrum, именно эти подпрограммы сделают это возможным.

В следующей главе Dark и STS расширяют этот математический фундамент до полноценной 3D-системы: метод средней точки для интерполяции вершин, отсечение задних граней и рендеринг закрашенных полигонов. Математика здесь — фундамент. Глава 5 — архитектура, построенная на нём.

Случайные числа: когда таблицы не помогают

Всё рассмотренное до сих пор в этой главе детерминировано. При одинаковых входных данных одно и то же умножение, один и тот же поиск синуса, одна и та же линия — дают одинаковый результат. Именно это и нужно для вращающегося каркасного куба или плавной плазмы.

Но иногда нужен хаос. Мерцающие звёзды в звёздном поле. Частицы, разлетающиеся от взрыва. Шумовые текстуры для генерации ландшафта. Перемешанный порядок для загрузочных экранов. В соревнованиях по sizecoding (256 байт или меньше) хороший генератор случайных чисел может создавать удивительно сложные визуальные эффекты почти из ничего.

У Z80 нет аппаратного генератора случайных чисел. Приходится синтезировать случайность из арифметики, и качество этой арифметики важнее, чем может показаться.

Трюк с регистром R

У Z80 есть встроенный источник энтропии, к которому многие кодеры обращаются в первую очередь: регистр R. Он автоматически инкрементируется при каждом цикле выборки инструкции (каждый цикл M1), перебирая значения 0-127. Прочитать его можно за 9 тактов:

```
ld a, r ; 9 T -- read refresh counter
```

Это *не* ГПСЧ. Регистр R полностью детерминирован — он увеличивается на единицу с каждой инструкцией, и его значение в любой точке определяется исключительно путём исполнения кода с момента сброса. В демо с фиксированным главным циклом R выдаёт одну и ту же последовательность каждый раз. Зато он полезен как источник начального значения (seed): прочитай R один раз при запуске (когда тайминг зависит от того, сколько пользователь ждал перед нажатием клавиши) и используй это непредсказуемое значение для инициализации настоящего ГПСЧ.

Некоторые кодеры подмешивают R в свой генератор при каждом вызове, добавляя настоящую энтропию от тайминга инструкций. Генератор Ion ниже использует именно этот трюк.

Четыре генератора от сообщества

В 2024 году Gogin (из российской ZX-сцены) собрал коллекцию подпрограмм ГПСЧ для Z80 и поделился ими для оценки. Gogin протестировал их систематически, заполняя большие растровые изображения для выявления статистических паттернов. Результаты поучительны — не все «случайные» подпрограммы одинаково случайны.

Вот четыре генератора из этой коллекции, упорядоченные от лучшего к худшему качеству.

CMWC-генератор Patrik Rak (Raxoft) (лучшее качество)

Это генератор **Complement Multiply-With-Carry** от Patrik Rak (Raxoft), использующий множитель 253 и кольцевой буфер из 8 байт. Математика CMWC хорошо изучена: Джордж Марсалья доказал, что определённые комбинации множителя и буфера дают последовательности с огромными периодами. С множителем 253 и буфером размера 8 теоретический период составляет $(253^8 - 1) / 254$ — примерно 2^{66} значений до повторения.

```
; 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

```

Алгоритм умножает текущий элемент буфера на 253, прибавляет значение переноса, сохраняет новый перенос и дополняет результат. Кольцевой буфер из 8 байт означает, что пространство состояний генератора огромно — 8 байт буфера плюс 1 байт переноса плюс индекс дают куда больше внутреннего состояния, чем может достичь любой одно-регистровый генератор.

Вердикт Gogin'a: **лучшее качество** в коллекции. При заполнении раstra 256x192 видимых паттернов не возникает даже в крупном масштабе.

Ion Random (второе место)

Изначально из Ion Shell для калькулятора TI-83, адаптирован для Z80. Этот генератор подмешивает регистр R в цикл обратной связи, достигая удивительно хорошей случайности всего в ~ 15 байтах:

```

; 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
```

Инъекция регистра R означает, что этот генератор даёт разные последовательности в зависимости от контекста вызова — количество исполненных инструкций между вызовами влияет на R, который возвращается в состояние. Для главного цикла демо с фиксированным таймингом R изменяется предсказуемо, но нелинейное смешивание (ADD + XOR) всё равно даёт хороший результат. В игре, где пользовательский ввод меняет паттерн вызовов, вклад R добавляет настоящую непредсказуемость.

Вердикт Gogin'a: **второе место**. Очень компактный, хорошее качество для своего размера.

XORshift 16 бит (посредственный)

16-битный XORshift-генератор — адаптация для Z80 из известного семейства Марсальи:

```
; 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
    ld   a, l           ; 4 T
    rra
    xor  h              ; 4 T
    ld   h, a           ; 4 T
    ld   a, l           ; 4 T
    rra
    ld   a, h           ; 4 T
    rra
    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-генераторы быстры и просты, но при всего 16 битах состояния период составляет максимум 65 535. Что ещё хуже, паттерн ротации битов создаёт видимые диагональные полосы при отображении на пиксели. Для быстрого звёздного поля или эффекта частиц это может быть приемлемо. Для заполнения больших областей экрана «шумом» паттерны становятся очевидными.

Вариант CMWC от Patrik Rak (посредственный)

Второй вариант CMWC от Patrik Rak (Raxoft), схожий по принципу с его версией выше, но с другой организацией буфера. Gogin обнаружил, что он создаёт **видимые паттерны в крупном масштабе** — вероятно, из-за взаимодействия распространения переноса с индексацией буфера. Мы включаем его в компилируемый пример (`examples/prng.a80`) для полноты, но для продакшена его версия с 8-байтным буфером выше строго лучше.

Подход Tribonacci из Elite

Заслуживает краткого упоминания: легендарная *Elite* (1984) использовала последовательность, подобную Трибоначчи, для процедурно генерируемой галактики. Три регистра замыкаются друг на друга в цикле обратной связи, создавая детерминированные, но хорошо распределённые последовательности. Ключевая идея — воспроизводимость: при одном и том же начальном значении генерируется одна и та же галактика, что позволяло всей вселенной «поместиться» в нескольких байтах состояния генератора. Дэвид Брэбен и Йен Белл использовали это для генерации 8 галактик по 256 звёздных систем из горстки байтов-seeds. Техника ближе к хэш-функции, чем к ГПСЧ, но принцип — малое состояние, большая видимая сложность — тот же, что движет демосценовым sizecoding.

Генератор галактик Elite: подробнее

Подход Tribonacci заслуживает более детального рассмотрения, поскольку иллюстрирует ключевой принцип: **ГПСЧ — это не просто источник случайных чисел, это алгоритм сжатия**.

Дэвиду Брэбену и Йену Беллу нужны были 8 галактик по 256 звёздных систем, каждая со своим названием, позицией, экономикой, типом правления и уровнем технологий. Хранить всё это явно потребовало бы килобайты. Вместо этого они хранили лишь 6-байтовый seed на галактику и детерминированный генератор, который разворачивал каждый seed в полные данные звёздных систем. Генератор был циклической обратной связью из трёх регистров — каждый шаг вращает и XOR'ит три 16-битных значения:

```
; 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
```

На Z80 тот же принцип работает с тремя регистровыми парами. Операция «twist» выдаёт детерминированные, но хорошо распределённые значения. Ключевое свойство: при одном и том же seed генерируется одна и та же галактика. Навигация между звёздами — это просто пересев и перегенерация.

Эта идея — **малое состояние, большая видимая сложность** — движет и демосценовым sizecoding. 256-байтное интро, заполняющее экран замысловатыми паттернами, делает ровно то же, что делала Elite: разворачивает крошечный seed в большой, сложный результат через детерминированный процесс.

Формирование случайности

Иногда нужны числа случайные, но следующие определённому распределению. Равномерный ГПСЧ даёт каждому значению равную вероятность, но реальные явления редко бывают равномерными: частота появления врагов, скорости частиц, высоты ландшафта — всё стремится кластеризоваться вокруг предпочтительных значений.

Распространённые трюки на Z80:

- **Треугольное распределение** — сложи два равномерных случайных байта и сдвинь вправо. Сумма концентрируется вокруг центра (128), создавая «естественно выглядящую» вариацию. Стоимость: два вызова ГПСЧ + ADD + SRL = ~20 лишних тактов.

```
; 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
```

- **Метод отклонения** — генерируешь случайное число, отбрасываешь значения вне нужного диапазона. Для диапазонов, равных степеням двойки, это бесплатно (просто AND с маской). Для произвольных диапазонов — цикл, пока значение не попадёт.
- **Взвешенные таблицы** — храни 256-байтную таблицу подстановки, где каждое выходное значение встречается пропорционально желаемой вероятности. Индексируй равномерным случайнм байтом. Таблица стоит 256 байт, но поиск мгновенен (7 тактов). Идеально, когда распределение сложное и фиксированное.
- **ГПСЧ как хэш-функция** — пропусти структурированные данные (координаты, номера кадров) через ГПСЧ, чтобы получить детерминированный шум. Именно так работают плазма и шумовые текстуры в sizecoding: random(x XOR y XOR frame) даёт разное на вид значение для каждого пикселя каждого кадра, но оно полностью воспроизводимо.

Seeds и воспроизводимость

В демо воспроизводимость обычно желательна: эффект должен выглядеть одинаково при каждом запуске, потому что кодер выстроил визуальный ряд под музыку. Инициализируй ГПСЧ один раз фиксированным значением — и последовательность будет детерминированной.

В игре важна непредсказуемость. Распространённые стратегии инициализации:

- **Системная переменная FRAMES (\$5C78)** — ПЗУ Spectrum поддерживает 3-байтный счётчик кадров по адресу \$5C78, который инкрементируется каждую 1/50 секунды от включения. Чтение даёт зависящий от времени seed, варьирующийся в зависимости от того, сколько машина работает. Art-top (Artem Topchiy) рекомендует использовать его для инициализации таблицы CMWC Patrik Rak:

```

; 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

```

- **Чтение R в момент пользовательского ввода** — точное число инструкций между сбросом и нажатием клавиши варьируется от запуска к запуску. LD A,R в этот момент захватывает энтропию тайминга.
- **Накопление счётчика кадров** — XOR регистра R в аккумулятор каждый кадр во время экрана-заставки; использовать накопленное значение как seed при старте игры.
- **Комбинация нескольких источников** — XOR вместе R, младший байт FRAMES и байт с «плавающей шиной» (на 48K Spectrum чтение определённых портов возвращает то, что ULA в данный момент читает из ОЗУ — источник позиционной энтропии).

Для демо просто инициализируй состояние генератора известным значением и оставь как есть. Компилируемый пример (examples/prng.a80) показывает все четыре генератора с фиксированными seeds.

Сравнительная таблица

Алгоритм	Размер (байт)	Скорость (такты)	Качество	Период	Приме- чания
Patrik Rak CMWC	~30 + 8 таблица	~170	Отлич- ное	$\sim 2^{66}$	Луч- ший в целом; 8- байтный буфер
Ion Random	~15	~75	Хорошее	Зависит от R	Ком- пакт- ный; подме- шива- ет ре- гистр R

Алгоритм	Размер (байт)	Скорость (такты)	Качество	Период	Приме- чания
XORshift 16	~25	~90	Посред- ственное	65 535	Види- мые диаго- наль- ные паттер- ны
Patrik Rak CMWC (alt)	~35 + 10 таблица	~180	Посред- ственное	$\sim 2^{66}$	Пат- терны видны в круп- ном мас- штабе
LD A,R в одиночку	2	9	Плохое	128	НЕ ГПСЧ; ис- поль- зуй только как seed

Для большинства демосценовых задач **CMWC Patrik Rak** — безоговорочный победитель: отличное качество, разумный размер и период настолько длинный, что он никогда не повторится за время демо. Если критичен размер кода (sizecoding, 256-байтные интро), **Ion Random** упаковывает замечательное качество в 15 байт. XORshift — запасной вариант, когда нужно что-то быстро и неважно визуальное качество.

Авторы: Коллекция ГПСЧ, оценка качества и растровое тестирование — **Gogin**. CMWC-генератор Patrik Rak основан на теории Complementary Multiply-With-Carry Джорджа Марсалы. Ion Random берёт начало из **Ion Shell** для калькулятора TI-83.

Все подсчёты тактов в этой главе — для тайминга Pentagon (без wait-состояний). На стандартном 48K Spectrum или Scorpion со спорной памятью ожидай более высокие значения для кода, исполняемого в нижних 32 КБ ОЗУ. См. Приложение А для полной справки по таймингу.

Источники: Dark / X-Trade, «Programming Algorithms» (Spectrum Expert #01, 1997); Gogin, коллекция ГПСЧ и оценка качества; Patrik Rak (Raxoft), CMWC-генератор; Ped7g (Peter Helcmanovsky), выявление пробела в знаковой арифметике и реview



Рис. 12: Вывод ГПСЧ — случайные цвета атрибутов заполняют экран, выявляя статистическое качество генератора

Глава 5: 3D на 3.5 МГц

«Вычисляй только то, что необходимо. Остальное выводи.» — Dark & STS, Spectrum Expert #02 (1998)

Предыдущая глава дала тебе строительные блоки: умножение, деление, таблицы синусов, рисование линий. Теперь мы собираем их вместе. Цель — вращающийся трёхмерный объект с закрашенными полигонами на ZX Spectrum: заполненные грани, отсечение задних граней, корректная сортировка по глубине — на приемлемой частоте кадров.

Именно здесь ты упираешься в стену.

Проблема: двенадцать умножений на вершину

Вращение точки в трёхмерном пространстве вокруг всех трёх осей требует серии тригонометрических умножений. Если вращать последовательно — сначала вокруг Z, затем Y, затем X — каждая ось требует четырёх умножений и двух сложений для преобразования двух координат. Три оси, четыре умножения каждая: двенадцать умножений на вершину.

Возьмём умножение сдвигом-и-сложением из Главы 4, которое стоит примерно 200 тактов (T-state). Двенадцать таких дают 2 400 тактов на вращение одной вершины. У простого куба 8 вершин: 19 200 тактов только на вращение. Глава 4 показала, что это умещается в бюджет кадра — едва-едва.

Теперь попробуй что-нибудь посложнее. Сферу, аппроксимированную 20 вершинами и 36 гранями:

Это 67% бюджета кадра Pentagon в 71 680 тактов, потраченные до того, как ты нарисовал хоть один пиксель. Ещё нужна перспективная проекция, отсечение задних граней, сортировка полигонов и собственно заливка. Места нет. Объект не может быть сложнее куба, если не найти принципиально более дешёвый способ вычисления позиций вершин.

Dark и STS его нашли.

Метод средней точки

Суть — в геометрии. Не каждая вершина объекта несёт независимую информацию. Многие вершины находятся в структурно предсказуемых позициях — средних точках рёбер, центрах граней, отражениях других вершин. Если выразить эти связи явно, можно заменить дорогие умножения дешёвым усреднением.

Куб как основа

Рассмотрим куб, центрированный в начале координат. У него 8 вершин, но это не 8 независимых точек. Это 4 пары диаметрально противоположных вершин. Если знаешь одну вершину пары, другая — её отрицание через центр:

$$\begin{array}{lll} v1 = (x, y, -z) & \rightarrow & v6 = (-x, -y, z) \\ v2 = (x, -y, z) & \rightarrow & v5 = (-x, y, -z) \\ v3 = (x, -y, -z) & \rightarrow & v4 = (-x, y, z) \end{array}$$

Поверни 4 вершины полной процедурой с 12 умножениями. Отрицай их, чтобы получить остальные 4. Отрицание на Z80 — это NEG — 8 тактов на одну координату, 24 такта на все три. Сравни с 2 400 тактами полного вращения. Ты сократил вычисление вершин почти вдвое.

Но метод средней точки идёт гораздо дальше отражения.

Вывод вершин усреднением

Ключевая операция — среднее: имея две уже вычисленные точки, их средняя точка — просто среднее их координат.

На Z80 это сложение и сдвиг:

```
; 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) сохраняет знаковый бит, поэтому работает корректно для отрицательных координат. Для всех трёх координат (x, y, z) усреднение стоит 36 тактов на производную вершину. Сравни с 2 400 тактами полного вращения.

Соотношение: усреднение в **66 раз дешевле** вращения.

Это означает, что можно строить сложные объекты из небольшого набора «базисных» вершин, которые врачаешь полностью, а затем выводишь все остальные вершины через цепочки усреднений. Чем больше вершин можно вывести, тем больше времени экономишь.

Построение сложных объектов

Допустим, тебе нужен объект из 20 вершин. С методом средней точки:

1. Полное вращение 4 базисных вершин: $4 \times 2\ 400 = 9\ 600$ тактов
2. Зеркальное отражение 4 вершин: $4 \times 24 = 96$ тактов
3. Вывод 12 вершин усреднением: $12 \times 36 = 432$ такта
- 4. Итого: 10 128 тактов**

Без метода средней точки те же 20 вершин стоили бы 48 000 тактов. Ты сэкономил 37 872 такта — более половины бюджета кадра высвобождено для проекции, отсечения и рендеринга.

Ограничение — топологическое: можно вывести вершину усреднением, только если она действительно лежит в средней точке двух других вершин (или достаточно близко, чтобы ошибка была невидима при разрешении 256x192). Это определяет, как ты проектируешь 3D-модели. Ты не моделируешь свободно, а потом оптимизируешь — ты проектируешь модель *вокруг структуры средних точек с самого начала*.

Dark и STS приводят примеры цепочек вывода:

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

Заметь, что v11 выведена из v8 и v9, которые сами являются производными. Цепочки могут уходить на несколько уровней вглубь. Каждый уровень добавляет лишь 36 тактов на вершину, так что стоимость остаётся пренебрежимой независимо от глубины.

Виртуальный процессор

Здесь Dark делает нечто, что кажется анахронизмом для 1998 года. Вместо жёсткого кодирования цепочек вывода для каждого конкретного объекта он проектирует крошечный интерпретатор — виртуальный процессор, — который исполняет «программы», описывающие, как вычислять вершины.

Архитектура

Виртуальный процессор имеет:

- **Один регистр** (рабочий регистр, хранящий одну 3D-точку — три байта: x, y, z)
- **64 ячейки ОЗУ** (каждая ячейка хранит одну 3D-точку — всего 192 байта)
- **4 инструкции**

Опкод	Биты	Имя	Операция
00	00nnnnnn	Load	регистр <- ячейка[n]
01	01nnnnnn	Store	ячейка[n] <- регистр
10	10nnnnnn	Average	регистр <- (регистр + ячейка[n]) / 2

Опкод	Биты	Имя	Операция
11	11-----	End	остановка выполнения

Каждая инструкция закодирована в одном байте: 2 бита на опкод, 6 бит на номер ячейки (0-63). Весь набор инструкций умещается в 256 возможных значений.

Исполнение

Цикл интерпретатора компактен:

```
; 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
```

Инструкция **Load** копирует значения x, y, z ячейки в рабочий регистр. **Store** копирует рабочий регистр обратно в ячейку. **Average** складывает координаты ячейки с рабочим регистром и сдвигает каждый результат вправо на один бит — операция средней точки. **End** завершает программу.

Написание программ

Цепочка вывода вершин становится простой последовательностью байтов. Dark использует компактную нотацию в статье:

```
; 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)
```

Нотация 128+5 кодирует %10000101 — опкод 10 (Average) с номером ячейки 5. 64+8 кодирует %01001000 — опкод 01 (Store) с номером ячейки 8. Чистые числа, упакованные в байты данных, формирующие крошечную предметно-ориентированную программу.

Полное описание объекта может выглядеть так:

```
; 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)
```

Тринадцать байт описывают вычисление четырёх производных вершин. Виртуальный процессор исполняет их примерно за $13 \times 30 = 390$ тактов (каждая инструкция занимает приблизительно 25–35 тактов в зависимости от типа). Четыре полностью повёрнутые вершины стоили бы 9 600 тактов. Экономия огромна.

Зачем виртуальный процессор?

Можно спросить: зачем не написать код усреднения прямо на Z80-ассемблере? Встроить сложения и сдвиги, пропустить накладные расходы интерпретатора. Это было бы чуть быстрее на вершину.

Ответ — гибкость. Виртуальный процессор отделяет *описание топологии* объекта от *исполнения вычисления* вершин. Сменить объект? Напиши новую программу — новую последовательность байтов данных. Код интерпретатора остается тем же. Можно хранить программы для нескольких объектов и переключаться между ними без затрат на код. Можно даже генерировать программы алгоритмически.

По сути, это предметно-ориентированный интерпретатор байткода — паттерн, который современные программисты узнают из игровых движков, шейдерных компиляторов и скриптовых языков. Dark спроектировал его в 1998 году, на ZX Spectrum, чтобы сэкономить такты на вычислении вершин. Архитектура чиста.

Вращение

С методом средней точки, обрабатывающим большинство вершин, тебе всё ещё нужно правильно вращать базисные вершины. Dark и STS используют последовательное вращение вокруг трёх осей в порядке: Z, затем Y, затем X. Каждое вращение использует таблицы синуса и косинуса из Главы 4.

Вращение вокруг оси Z

Вращение вокруг Z влияет только на X и Y:

$$Y' = -X * \sin(Az) + Y * \cos(Az)$$

На Z80-ассемблере, используя знаковое умножение 8x8 и 256-элементные таблицы синуса/косинуса:

```
; 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

```

Тот же паттерн повторяется для вращения вокруг оси Y (затрагивая X и Z) и вокруг оси X (затрагивая Y и Z). Dark обворачивает все три в единую процедуру ROTATE, которая принимает три параметра угла и преобразует точку на месте.

Обрати внимание на деталь о сохранении исходного значения X. Вторая формула использует X до вращения, а не только что вычисленный X'. Типичная ошибка — использовать уже обновлённую координату, что даёт искажённое вращение. Dark явно говорит об этом в статье.

Стоимость на базисную вершину

Каждое вращение по оси требует 4 умножения и 2 сложения. При 200 тактах на умножение и 11 тактах на 16-битное сложение:

Three axes: $3 \times 822 = 2,466$ T-states per vertex

С 4 базисными вершинами: примерно 9 864 такта на вращение. Добавь исполнение программы средней точки — и ты получаешь полное вычисление вершин для произвольно сложного объекта за долю наивной стоимости.

Проекция

Когда все вершины повёрнуты в 3D-пространстве, нужно спроектировать их на 2D-экран.

Параллельная проекция

Простейший подход: полностью игнорировать координату Z. Просто использовать X и Y как экранные координаты (со сдвигом для центровки объекта).

```

; 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

```

Стоимость: практически нулевая. Результат выглядит плоско — объекты не кажутся уходящими вдаль. Параллельная проекция полезна для каркасных

превью и эффектов, где само вращение создаёт иллюзию глубины, но ей не хватает убедительности перспективы.

Перспективная проекция

Перспектива делает ближние объекты крупнее, а дальние — мельче, создавая ощущение глубины, которое делает 3D убедительным:

```
Yscreen = (Y * Scale) / (Z + Zdistance) + Yoffset
```

Scale контролирует угол обзора. Zdistance — расстояние от камеры до плоскости проекции — предотвращает деление на ноль, когда Z приближается к камере, и управляет интенсивностью масштабирования по глубине. Xoffset и Yoffset центрируют проекцию на экране.

Дорогая операция здесь — деление. Одно деление на координату, две координаты на вершину. С логарифмическим делением из Главы 4 (~60 тактов на деление) стоимость умеренна:

```
; 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              ; take high byte as numerator
    call  log_divide        ; A = A / C
    add   a, YOFFSET
    ld    (screen_y), a

ret
```

Каждая вершина стоит два умножения (400 тактов) и два логарифмических деления (120 тактов), плюс накладные расходы — примерно 600 тактов на вершину. Для 20 вершин: 12 000 тактов. В сочетании с вращением методом средней точки мы оказываемся примерно на 22 000 тактах за все вычисления вершин и проекцию. Меньше трети бюджета кадра.

Закрашенные полигоны

Каркасный объект — это набор рёбер. Закрашенный объект — набор заполненных полигонов. Переход от каркаса к сплошному объекту требует трёх дополнительных возможностей: определение видимых граней, сортировка по глубине и заливка пикселями.

Отсечение задних граней

У замкнутого 3D-объекта есть грани, обращённые к зрителю, и грани, обращённые от него. Задние грани скрыты и не нуждаются в рисовании. Их пропуск экономит время рендеринга и создаёт корректный сплошной вид без полного буфера глубины.

Тест геометрический. Для каждой грани вычисляем Z-компоненту нормали поверхности через векторное произведение двух рёберных векторов:

Edge vectors:

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

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

Если результат положительный, грань обращена к зрителю — рисуем. Если отрицательный — грань отвёрнута, отбрасываем. Если ноль — грань видна с ребра и невидима.

```
; 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

```

Два умножения и вычитание на грань. При 400 тактах на умножения плюс накладные расходы, тест стоит примерно 500 тактов на грань. Для объекта с 12 гранями это 6 000 тактов — и за каждую отброшенную грань ты экономишь всю стоимость её заливки.

На типичном вращающемся объекте примерно половина граней в любой момент обращена от зрителя. Их отсечение вдвое снижает нагрузку заливки.

Сортировка по Z

Для выпуклого объекта (куб, тетраэдр) одного отсечения задних граней достаточно для корректного результата: каждая видимая грань полностью видна, перекрытий нет. Для невыпуклых или многообъектных сцен нужно рисовать грани в порядке от дальних к ближним, чтобы ближние перезаписывали дальние — алгоритм художника.

Dark и STS вычисляют значение глубины для каждой видимой грани (обычно среднее Z её вершин) и сортируют список граней соответственно. Простая сортировка вставками подходит для малого количества граней — сортировка 6-12 граней занимает пренебрежимое время по сравнению с их заливкой.

```

; 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
; ...

```

Заливка выпуклых полигонов

Когда известно, какие грани рисовать и в каком порядке, нужно их залить. Выпуклый полигон (все внутренние углы менее 180 градусов) можно залить простым построчным подходом:

1. Найди верхнюю и нижнюю вершины.
2. Двигайся вниз по левому и правому ребру одновременно, строка за строкой.
3. Для каждой строки рисуй горизонтальную линию от левого ребра до правого.

Проход по рёбрам использует инкрементальное шагание в стиле Брезенхэма — без деления на строку, только сложения и условные инкременты. Сама горизонтальная заливка — плотный цикл записи байтов:

```
; 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
```

Это упрощение — реальные заливщики полигонов должны обрабатывать неполные байты на левом и правом краях, где граница полигона проходит внутри байта, а не по его границе. Эти крайние случаи добавляют сложности, но не много стоимиости, поскольку встречаются лишь дважды на строку.

Собираем всё вместе

Полный цикл кадра для вращающегося 3D-объекта с закрашенными полигонами следует такой последовательности:

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)

Для объекта из 20 вершин и 18 граней с 4 базисными вершинами, бюджет

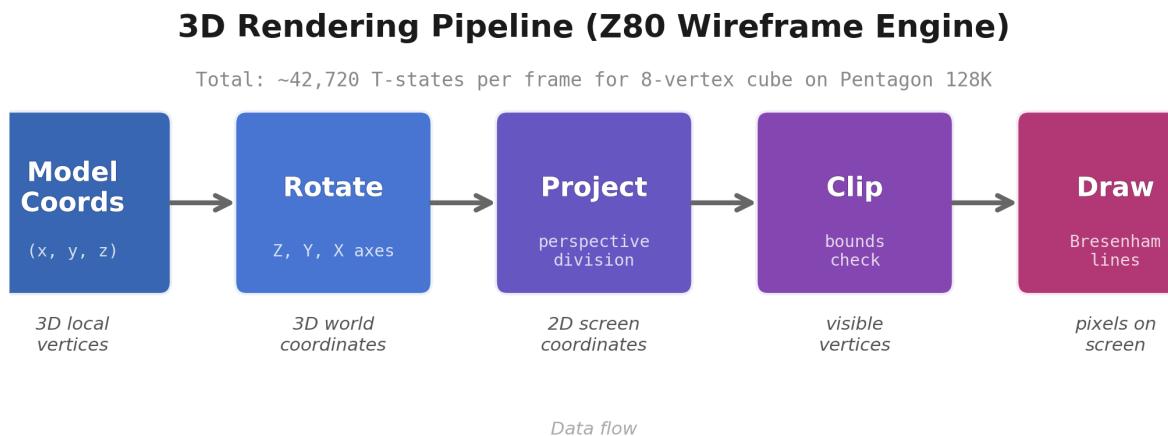


Рис. 13: 3D rendering pipeline: model, rotation, projection, screen

кадра распределяется так:

Этап	Вершин/Граней	Стоимость ед.	Итого
Вращение (базис)	4	2 466	9 864
Отрицание (зеркала)	4	24	96
Вывод средних точек	12	36	432
Проекция	20	600	12 000
Тест задних граней	18	500	9 000
Сортировка по Z	~9 видимых	-	~200
Заливка полигонов	~9 видимых	~1 500 сп.	~13 500
Итого			~45 092

Сорок пять тысяч тактов из 71 680 доступных. Плотно, но рабочо — остаётся 26 000 тактов на очистку экрана, обновление углов и рисование линий или контуров, которые придают объекту чёткость. И это для объекта из 20 вершин, куда более сложного, чем всё, что ты мог бы себе позволить при наивном вращении.

Форма объектов

Метод средней точки влияет на то, как ты думаешь о 3D-моделях. Ты не проектируешь полигональную сетку, а потом оптимизируешь — ты начинаешь с топологии, которую метод требует.

Хороший объект для метода средней точки начинается с малого базиса. Четыре полностью повёрнутые точки определяют тетраэдро-подобный скелет. Отрицание удваивает их до восьми. Усреднение средних точек заполняет остальное. Искусство — в выборе базисных вершин, дающих полезные производные точки.

Рассмотрим построение объекта из 14 вершин с нуля:

```
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
```

Программа виртуального процессора для этого — 19 байт:

```
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
```

Девятнадцать байт данных заменяют 14 400 тактов умножений при вращении 6 производных вершин.

Можно пойти дальше. Второй уровень усреднения (вывод из уже выведенных точек) не стоит ничего дополнительно на инструкцию — виртуальному процессору всё равно, содержит ячейка повёрнутую или производную точку. Dark и STS описывают цепочки глубиной три-четыре уровня, создавая объекты с 30 и более вершинами из всего 3-4 базисных точек.

Ограничение — точность. Каждый шаг усреднения вносит ошибку округления до 0.5 единицы (от целочисленного сдвига). После трёх уровней вывода кумулятивная ошибка может достичь 1.5 единицы — заметно на объекте размером 60 пикселей, невидимо на 120. Проектируй объекты достаточно крупными, чтобы округление было ниже разрешения экрана.

Практика: вращающийся закрашенный объект

Вот план построения полноценного вращающегося 3D-объекта с использованием всего из этой главы.

Шаг 1: Определение объекта

Начинаем с базисных вершин. Усечённый октаэдр хорошо подходит для метода средней точки:

```
; 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)
```

Шаг 2: Программа средней точки

```
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
```

Шаг 3: Определение граней

```
; 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
```

Шаг 4: Цикл кадра

```
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

```

Это каркас. Каждый `call` скрывает процедуру, построенную из техник этой главы и Главы 4. Сам цикл кадра чист — обновить, вычислить, нарисовать, повторить.

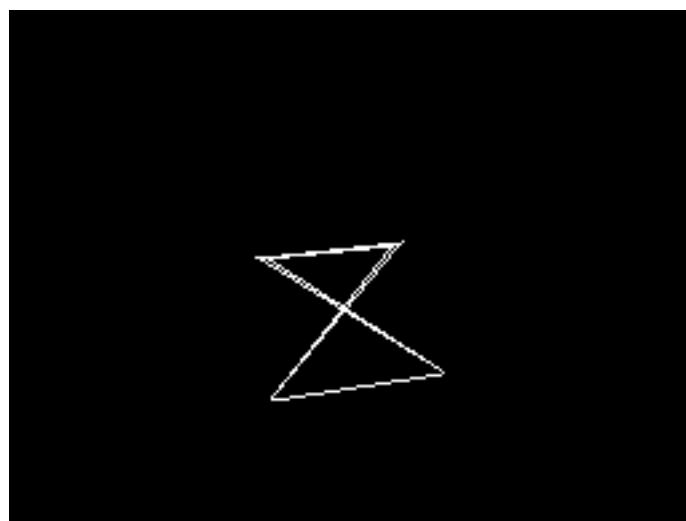


Рис. 14: Вращающийся каркасный куб, отрендеренный на ZX Spectrum методом средней точки и перспективной проекцией

Исторический контекст: от журнала к демо

Dark и STS опубликовали метод средней точки в *Spectrum Expert #02* в 1998 году. Они были молодыми кодерами в Санкт-Петербурге, писавшими для дискового журнала, распространявшегося внутри российского ZX Spectrum-сообщества. Статьи написаны в прямом, практическом стиле людей, обучающих своих коллег: вот проблема, вот трюк, вот код.

Но *Spectrum Expert* не был академическим упражнением. Dark был из X-Trade, той самой группы, что выпустила *Illusion* — демо, занявшее первое место на ENLiGHT'96. Алгоритмы в журнале — не теоретические предложения; это

строительные блоки реального, побеждавшего на соревнованиях демо-кода. Таблицы синусов из Главы 4 управляли ротозумером. Рисовальщик линий рендерил каркасы. А метод средней точки приводил в движение 3D-объекты.

Подход с виртуальным процессором особенно впечатляет в ретроспективе. В 1998 году доминирующие парадигмы в профессиональной разработке игр смещались в сторону аппаратного ускорения и прочь от программного рендеринга. На Spectrum аппаратного ускорения не существовало. Всё было программным, и программное обеспечение приходилось *проектировать* — не просто писать, а архитектурно выстраивать. Байткодовый интерпретатор Dark'a для вычисления вершин — архитектурное решение, которое не выглядело бы чуждо в анимационной системе или шейдерном компиляторе современного игрового движка. Он отделяет данные от исполнения, позволяет быстро итерировать над дизайном объектов и поддерживает компактность горячего цикла.

Связь с *Illusion* глубже общего авторства. Когда Introspec дизассемблировал *Illusion* двадцать лет спустя, он обнаружил ту же самую математическую инфраструктуру: умножитель сдвигом-и-сложением, параболическую таблицу синусов, делитель на логарифмических таблицах. Метод средней точки и виртуальный процессор — расширения этой инфраструктуры — то же мышление, применённое к другой задаче. Dark не просто публиковал алгоритмы; он документировал инженерную философию, стоящую за его собственным демо-победителем.

В следующей главе мы вблизи рассмотрим один из самых впечатляющих эффектов *Illusion*: текстурированную сферу. Она использует те же таблицы синусов и ту же арифметику с фиксированной точкой из Главы 4, в сочетании с техниками самомодифицирующегося кода из Главы 3 и совершенно иным подходом к задаче рендеринга. Метод средней точки и сфера — родственники, рождённые одним кодером, одними инструментами, одним неослабевающим стремлением уместить невозможное в 71 680 тактов.

Итого

- Наивное вращение 3D-объектов требует 12 умножений на вершину — слишком дорого для сложных объектов на 3.5 МГц Z80.
- **Метод средней точки** полностью вращает лишь несколько базисных вершин, а остальные выводят через усреднение. Усреднение стоит ~36 тактов на вершину против ~2 400 при полном вращении — в 66 раз дешевле.
- **Виртуальный процессор** с 4 инструкциями (Load, Store, Average, End) исполняет компактные «программы», описывающие цепочки вывода вершин. Топология объекта — это данные, а не код.
- **Вращение** использует последовательные Z/Y/X-преобразования с таблицами синуса/косинуса из Главы 4.
- **Перспективная проекция** использует логарифмическое деление из Главы 4 для деления на Z.
- **Отсечение задних граней** через тест нормали векторным произведением устраниет невидимые грани по ~500 тактов каждая.
- **Сортировка по Z** с алгоритмом художника обрабатывает перекрывающиеся грани для невыпуклых объектов.

- Закрашенный объект из 20 вершин можно отрендерить за ~45 000 тактов на кадр — плотно, но выполнимо в рамках бюджета в 71 680 тактов.
- Эти техники были опубликованы в Spectrum Expert #02 (1998) той же командой, что создала *Illusion*. Журнальные статьи документируют инженерию, стоящую за демо.

*Все подсчёты тактов в этой главе — для тайминга *Pentagon* (без wait-состояний). На стандартном 48K Spectrum со спорной памятью ожидай более высокие значения для кода, исполняемого в нижних 32 КБ ОЗУ. См. Приложение А для полной справки по таймингу.*

Источники: Dark & STS, «Программирование: 3D-графика» (Spectrum Expert #01, 1997); Dark & STS, «Программирование: 3D-графика — Метод средней точки» (Spectrum Expert #02, 1998). Конструкция виртуального процессора и примеры вывода средних точек взяты непосредственно из статьи SE#02.

Глава 6: Сфера — Текстурный маппинг на 3.5 МГц

«Кодерские эффекты — это всегда эволюция вычислительной схемы.» — Introspec, 2017

1996 год, и демо под названием *Illusion* занимает первое место на ENLiGHT'96 в Санкт-Петербурге. Зрители наблюдают, как монохромное изображение обогащается вокруг вращающейся сферы, плавно вращаясь в реальном времени, на ZX Spectrum, работающем на 3.5 МГц без какого-либо аппаратного ускорения. Ни блиттера. Ни GPU. Ни сопроцессора. Только Z80, 48 килобайт непрерывного ОЗУ и всё, что двадцатилетний кодер по имени Dark мог из них выжить.

Двадцать лет спустя, в марте 2017 года, Introspec садится с копией бинарника и дизассемблером. Он разбирает цикл рендеринга инструкцию за инструкцией, считает такты (T-state), сопоставляет адреса памяти со структурами данных и публикует свои находки на Нуре. Далее следует один из самых детальных публичных разборов демосценового эффекта, когда-либо написанных для ZX Spectrum — и в ветке комментариев, разгоревшейся под статьёй, дискуссия о том, что действительно важно в рендеринге реального времени на ограниченном оборудовании.

Эта глава следует анализу Introspec'a. Мы заглянем ему через плечо, пока он трассирует код, поймём, почему сфера работает именно так, а затем построим упрощённую версию сами.

Проблема: круглый объект на квадратном экране

Сфера на экране — не сфера. Это круг, заполненный искажённым изображением. Искажение следует правилам сферической проекции: пиксели вблизи экватора расположены равномерно, пиксели вблизи полюсов сжаты по горизонтали, и всё отображение изгибаются, создавая иллюзию трёхмерной поверхности.

Исходное изображение в *Illusion* хранится как монохромная битовая карта — один байт на пиксель, где каждый байт равен либо 0, либо 1. По стандартам Spectrum это расточительно, где экранная память упаковывает восемь пикселей в байт, но это даёт существенное преимущество: рендерящий код может обращаться с пикселями как с арифметическими значениями, а не с битовыми позициями.

Задача, таким образом, такова: читать пиксели из исходного изображения, выбирать их согласно сферической проекции, упаковать восемь из них в один экранный байт и записать этот байт в видеопамять. Сделать это для каждого видимого байта сферы. Сделать достаточно быстро для анимации. Сделать на Z80 частотой 3.5 МГц.

Ключевая идея: код, который пишет код

Первый вопрос, который задаёт каждый Z80-программист: как выглядит внутренний цикл? На машине, где один NOP занимает 4 такта и бюджет кадра составляет примерно 70 000 тактов, внутренний цикл *и есть* программа. Всё остальное — инициализация, генерация таблиц, управление кадрами — это накладные расходы, выполняемые однократно или редко. Внутренний цикл работает тысячи раз за кадр.

Решение Dark'a — не иметь фиксированного внутреннего цикла вообще.

Вместо этого рендерящий код *генерируется во время выполнения*. Для каждой горизонтальной линии сферы программа конструирует последовательность Z80-инструкций, настроенных под геометрию этой линии. Сгенерированный код читает исходные пиксели по порядку, накапливает их в экранные байты через сдвиги и сложения и продвигается по исходным данным на расстояния, меняющиеся с кривизной сферы. Разные линии сферы дают разный код.

Это техника, встречающаяся повсюду на демосцене: самогенерируемый код, иногда называемый «скомпилированные спрайты» при применении к рендерингу спрайтов, или «генерация развёрнутых циклов» в общем случае. Что делает версию для сферы отличительной — вариативность. Скомпилированный спрайт фиксирован — однажды сгенерированный, он рисует одну и ту же форму каждый раз. Код сферы меняется с углом вращения, потому что разные исходные пиксели становятся видны по мере поворота сферы.

Внутри дизассемблера

Introspec отследил рендерящий движок до блока сгенерированного кода и набора таблиц подстановки, начинающихся с адреса \$6944. Таблицы кодируют геометрию сферы как серию *расстояний пропуска*: для каждой позиции вдоль строки развёртки сферы — сколько исходных пикселей пропустить перед выборкой следующего.

На экваторе расстояния пропуска примерно одинаковы — исходное изображение отображается на сферу с минимальным искажением. Вблизи полюсов горизонтальное сжатие проекции означает большие пропуски между выбираемыми пикселями. На самом верху и низу видимы лишь несколько пикселей на линию, и пропуски могут быть значительными.

Сгенерированный внутренний цикл имеет повторяющуюся структуру. Для каждого экранного байта (восемь упакованных пикселей) он исполняет последовательность вроде этой:

```
; --- 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 ...

```

Ключевая деталь: между каждым `add a,(hl)` количество инструкций `inc l` варьируется. В одной позиции может быть один `inc l` (выборка соседних пикселей). В другой — три или четыре (пропуск сжатых областей проекции). Таблицы подстановки по адресу \$6944 кодируют точное количество `inc l` для каждой позиции.

Рассмотрим внимательнее, что происходит с одним пикселием:

```

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)

```

Это минимальная стоимость: 15 тактов на сдвиг аккумулятора и выборку одного пикселя, плюс 4 такта за каждый пропущенный исходный байт. После восьми таких последовательностей аккумулятор содержит полный экранный байт.

Обрати внимание, что указатель исходных данных продвигается через `inc l`, а не `inc hl`. Это намеренно. INC HL занимает 6 тактов; INC L — 4. Ограничивающая исходные данные рамками одной 256-байтной страницы (чтобы менялся только младший байт адреса), Dark экономит 2 такта на каждое продвижение. Когда делаешь это тысячи раз за кадр, эти 2 такта набегают.

Здесь есть тонкость, которую легко упустить. Исходное изображение хранится как один байт на пиксель, и INC L оборачивается в пределах 256-байтной страницы. Это значит, что каждая строка исходных данных должна умещаться в 256 байт, а буфер исходных данных должен быть выровнен по странице. Ограничение определяет всю раскладку памяти демо.

Подсчёт тактов

Introspec вычислил стоимость одного выходного байта:

101 + 32x тактов

где x — среднее количество дополнительных инструкций INC L на пиксель сверх обязательной одной. Проверим.

Фиксированная стоимость на пиксель:

Инструкция	Такты
add a,a	4
add a,(hl)	7
inc l	4
Промежуточный итог	15

Для 8 пикселей фиксированная стоимость — $8 \times 15 = 120$ тактов. Но есть дополнительные накладные расходы на байт: код должен записать собранный байт в экранную память и подготовиться к следующему. Допустим, последовательность вывода выглядит примерно так:

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

Также может быть обнуление аккумулятора (*xor a* или подобное) в начале каждого байта. Принимая измеренную Introspec'ом цифру в 101 такт как фиксированную базовую стоимость на байт, накладные расходы сверх чистой выборки пикселей составляют примерно $101 - 120 = \dots$ что означает, что базовая цифра уже включает инструкции вывода, и часть пиксельной работы чередуется иначе, чем наивный подсчёт предполагает.

Чистое прочтение формулы: 101 такт фиксированных накладных расходов (вывод, управление указателями, инициализация на байт), плюс 32 такта на каждый дополнительный пропуск. «32» возникает из 8 пикселей, умноженных на 4 такта за каждый дополнительный *INC L*, что даёт *x* как среднее количество дополнительных пропусков на позицию пикселя в этом байте. Когда сфера вблизи экватора, *x* мало — проекция близка к равномерной. Вблизи полюсов *x* велико, и рендеринг замедляется. Но полюса также требуют меньше байтов (сфера там уже), так что общая нагрузка примерно сбалансирована.

Достаточно ли это быстро? Кадр Spectrum — приблизительно 70 000 тактов (на Pentagon: 71 680). Сфера диаметром 56 пикселей занимает примерно 7 байт в самом широком месте. По всей высоте нужно отрендерить возможно 200-250 байт. При 101 такте на байт (экваториальном, *x* около нуля) это примерно 25 000 тактов — комфортно укладывается в бюджет одного кадра, с запасом на очистку экрана, поиск по таблицам и прочую бухгалтерию. Даже вблизи полюсов, где *x* может в среднем составлять 2-3, стоимость байта растёт до 165-197 тактов, но рисовать нужно меньше байт. Арифметика сходится. Укладывается.

Проход генерации кода

Перед работой внутреннего цикла его генерирует проход генерации кода. Этот проход читает таблицы подстановки по адресу \$6944, кодирующие геометрию сферы для текущего угла вращения, и выдаёт Z80-инструкции в буфер:

1. Для каждой строки развёртки сферы прочитать расстояния пропуска из таблицы.
2. Выдать *add a,a*, за ним *add a,(hl)* для каждого пикселя.
3. Выдать соответствующее количество инструкций *inc l* на основе расстояния пропуска.

4. После каждого 8 пикселей выдать инструкцию вывода для записи собранного байта в экранную память.
5. В конце каждой строки выдать возврат или переход к обработчику следующей строки.

Сгенерированный блок кода затем вызывается напрямую. Процессор исполняет инструкции как обычную подпрограмму, но они были написаны мгновением ранее генератором кода. Это самомодифицирующийся код в самом буквальном смысле — программа генерирует программу, которая рисует экран.

Проход генерации кода сам по себе не бесплатен, но он выполняется один раз за кадр (или один раз за шаг вращения), тогда как сгенерированный внутренний цикл работает сотни раз. Амортизированная стоимость пренебрежима.

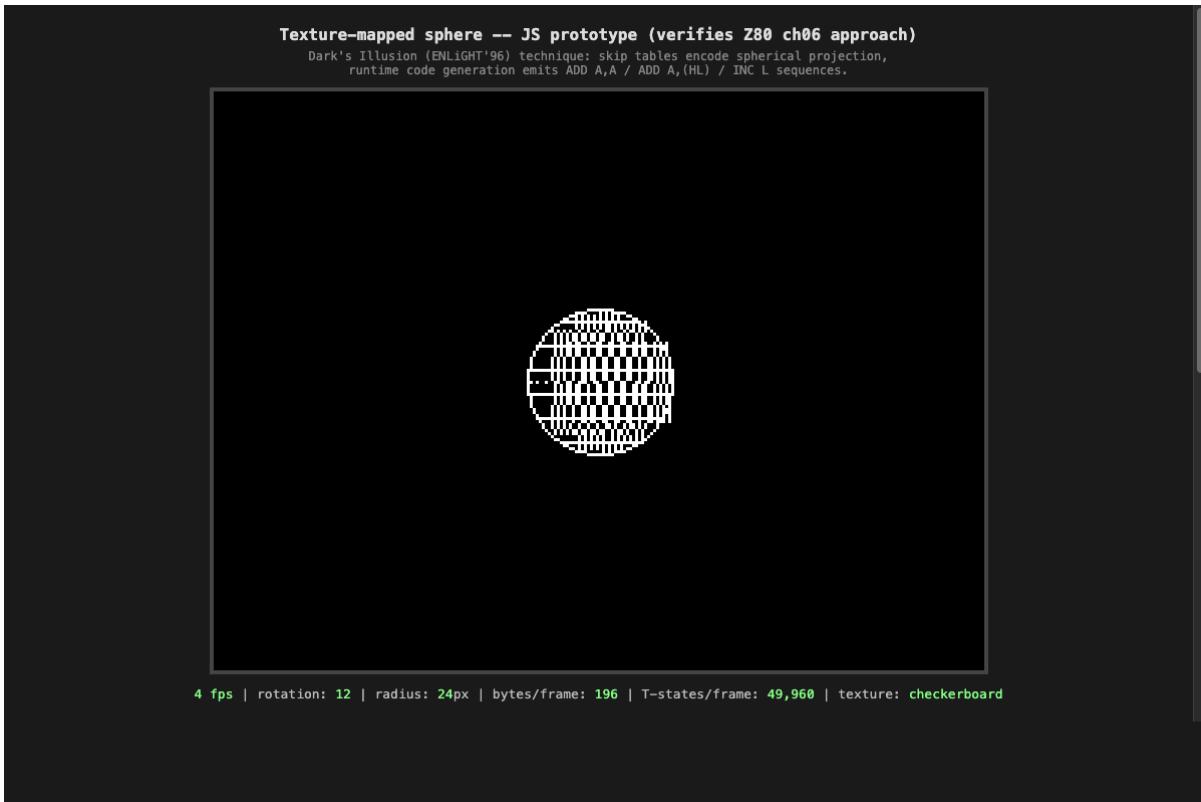


Рис. 15: Прототип текстурированной сферы — таблицы пропусков кодируют сферическую проекцию, генерация кода во время выполнения формирует последовательности пикселей

Что знал Dark: Spectrum Expert и строительные блоки

В этой истории есть деталь, превращающая её из технической диковинки в нарративную арку. Dark — кодер, стоящий за эффектом сферы в Illusion — это тот же Dark, который написал статью «Алгоритмы программирования» в Spectrum Expert #01, опубликованном в 1997 году.

Эти статьи охватывают умножение (сдвиг-и-сложение vs. поиск по таблице квадратов), деление (восстанавливающее и логарифмическое), генерацию таблиц синуса через параболическую аппроксимацию и рисование линий по Брезенхэму с оптимизированными матричными блоками 8x8. Это учебный

материал, написанный для сообщества программистов ZX Spectrum, объясняющий фундаментальные техники, которые нужны любому демо-кодеру.

И они, совершенно точно, являются строительными блоками, использованными в Illusion.

Сфера требует: тригонометрических таблиц подстановки для вычисления проекции (синус/косинус, параболическая аппроксимация из статьи Dark'a). Умножения с фиксированной точкой для масштабирования. Аккуратной раскладки памяти для скорости (та же дисциплина подсчёта тактов, которой Dark учит на протяжении всех статей). Подход с таблицами пропусков для кодирования геометрии сферы — прямое применение мышления на основе предвычислений, которое Dark пропагандирует.

Dark сначала написал демо — Illusion победил на ENLiGHT'96. Затем, в 1997–98 годах, он опубликовал учебник, в котором объяснил каждую использованную технику. Двадцать лет спустя Introspec провёл реверс-инжиниринг демо и нашёл именно те алгоритмы, которые Dark задокументировал. У нас есть обе стороны истории: практик, объясняющий свои методы постфактум, и аналитик, подтверждающий, что именно эти методы содержатся в готовом продукте.

Дебаты на Hype: внутренние циклы vs. математика

Статья Introspec'a 2017 года на Hype вызвала длинную ветку комментариев. Среди наиболее содержательных обменов была дискуссия между kotsoft и Introspec'ом о том, где лежит настоящая работа эффекта вроде этого.

kotsoft утверждал, что математический подход к проекции — то, как вычисляешь, какой исходный пиксель отображается на какую экранную позицию — является ключевым проектным решением. Ошибись в проекции, или используй наивный алгоритм, и никакая оптимизация внутреннего цикла тебя не спасёт. Математическая модель определяет, *осуществим ли вообще эффект на данном оборудовании*.

Introspec возражал, что внутренний цикл — это место, где такты реально расходуются. Можно иметь красивую математическую модель, но еслирендерящий код стоит 200 тактов на байт вместо 100, ты сократил частоту кадров вдвое. Математический подход определяет, что вычислять; внутренний цикл определяет, *можно ли вычислить это вовремя*.

Оба правы, и напряжение между ними высвечивает нечто фундаментальное в демосценовом кодировании. Демо-эффект — не чистая математика и не чистая инженерия. Это пересечение: элегантная вычислительная схема (проекция сферы, закодированная как таблицы пропусков), соединённая с эффективной стратегией исполнения (сгенерированные развёрнутые циклы с продвижениями INC L). По отдельности ни то, ни другое не достаточно.

Резюме Introspec'a схватывает суть: «кодерские эффекты — это всегда эволюция вычислительной схемы.» Ключевое слово — *эволюция*. Ты не берёшь алгоритм из учебника и оптимизируешь его, пока он не влезет. Ты эволюционируешь алгоритм и реализацию совместно, каждый ограничивает и обеспечивает другого, пока не найдёшь форму, работающую в рамках бюджета оборудования.

Практика: упрощённая вращающаяся сфера 56x56

Набросаем, как ты бы построил упрощённую версию этого эффекта. Целевая сфера — 56x56 пикселей — 7 байт в ширину на экваторе, 56 строк в высоту. Цель — не воспроизвести полноценный рендерящий движок Illusion, а понять ядро техники достаточно хорошо для реализации.

Шаг 1: Предвычисление геометрии сферы

Для каждой строки развёртки y (от -28 до +27, центрированной на сфере) вычисляем видимую дугу:

Это даёт полуширину сферы на этой строке. Для каждой позиции пикселя x в пределах дуги вычисляем соответствующие долготу и широту на поверхности сферы:

```
longitude = arcsin(x / radius_at_y) + rotation_angle
```

Это даёт координаты (u, v) в исходной текстуре для каждого экранного пикселя.

Шаг 2: Построение таблиц пропусков

Вместо хранения полных пар (u, v) для каждого пикселя (непомерно дорого по памяти) вычисляем *разницу в позиции источника* между соседними экранными пикселями. Для каждой строки нужен список значений пропуска: сколько исходных пикселей пропустить между последовательными экранными выборками.

Вблизи экватора последовательные экранные пиксели соответствуют почти соседним исходным пикселям — пропуски равны 1. Вблизи полюсов проекция сжимает, и ты пропускаешь больше исходных пикселей — пропуски 2, 3 или более.

Сохраняем это как таблицу. Для нашей сферы 56x56 нужно максимум 56 записей на строку (самая широкая), умножить на 56 строк, умножить на один байт на запись. Это максимум 3 136 байт для одного угла вращения — но на практике можно использовать вертикальную симметрию (верхняя половина зеркальна нижней) и хранить только половину таблицы.

Для анимации нужны таблицы пропусков для нескольких углов вращения. С 32 шагами вращения таблицы заняли бы $32 \times 1\ 568 =$ около 49 КБ. Это переполняет доступную память, поэтому на практике используешь меньше шагов, грубее угловое разрешение или перегенерируешь таблицы на лету из компактного представления.

Шаг 3: Генерация рендерящего кода

Для каждого кадра читаем таблицу пропусков текущего угла и генерируем Z80-код:

```
; 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

```

Это упрощение — реальный код Illusion более плотно интегрирован, и Dark,

вероятно, использовал более компактный и эффективный генератор кода. Но принцип тот же: читать расстояния пропусков, выдавать опкоды.

Шаг 4: Исполнение и отображение

Когда буфер кода заполнен, вызываем его как подпрограмму:

```
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
```

Сгенерированный код проходит через всю сферу, читая исходные пиксели, упаковывая их в экранные байты и записывая в видеопамять. По возвращении сфера нарисована.

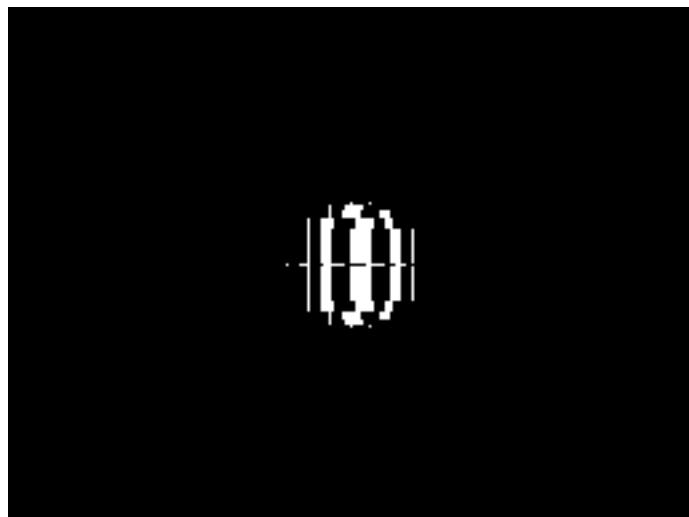


Рис. 16: Контур сферы, отрендеренный на ZX Spectrum — монохромная текстура наложена на вращающуюся сферу с помощью генерации кода по таблицам пропусков

Для анимации увеличиваем угол вращения, загружаем соответствующую таблицу пропусков (или перегенерируем её), перегенерируем код и рендерим заново.

Шаг 5: Раскладка исходного изображения

Исходная текстура должна быть организована для быстрого последовательного доступа. Поскольку рендерящий код использует INC L для продвижения, текстура должна быть выровнена по странице (начинаться по адресу, где младший байт равен \$00), и каждая строка должна умещаться в 256 байт. Текстура шириной 256 пикселей, хранящаяся как один байт на пиксель, идеально соответствует этому ограничению: каждая строка занимает одну страницу.

Для монохромного случая каждый пиксель — \$00 или \$01. Это значит, что ADD A,(HL) добавляет либо 0 (пиксель выключен), либо 1 (пиксель включен) к младшему биту аккумулятора, сразу после того как ADD A,A сдвинул всё вверх. Результат — побитно упакованный экранный байт, где каждый бит соответствует одному выбранному исходному пикслю.

Общий паттерн

Сфера в Illusion — конкретный экземпляр общего демосценового паттерна, встречающегося на протяжении всей этой книги. Паттерн состоит из трёх частей:

Предвычисление. Дорогостоящая математическая работа — проекция, тригонометрия, преобразования координат — выполняется однократно (или раз за кадр) и сохраняется в компактных таблицах. Таблицы кодируют, чторендерить, не кодируя как.

Генерация кода. Сам рендерящий код генерируется из таблиц. Это устраивает ветвления, счётчики циклов и условную логику из внутреннего цикла. Каждая инструкция в сгенерированном коде выполняет полезную работу. Нет накладных расходов на «выяснение, что делать дальше» — это решение было принято при генерации.

Последовательный доступ к памяти. Внутренний цикл читает данные последовательно, продвигая указатель однобайтными инкрементами. Это самый быстрый паттерн доступа на Z80, где косвенные загрузки через регистр (`LD A, (HL)`) дёшевы, а индексная адресация (`LD A, (IX+d)`) дорога.

Ротозумер в следующей главе использует тот же паттерн. Так же как и точечный скроллер в Главе 10. Так же как атрибутные туннели в Главе 9. Детали различаются — разные таблицы, разный сгенерированный код, разные форматы данных — но архитектура одна. Introspec понял это, когда написал, что кодерские эффекты — это «эволюция вычислительной схемы». Сфера, ротозумер, туннель: все они эволюционировали из одного фундаментального подхода. Эволюция — в деталях — какое вычисление, какая раскладка таблицы, какой внутренний цикл — но скелет общий.

Dark понимал это в 1996 году. Он закодировал это в своих статьях *Spectrum Expert* в 1997-м. Introspec подтвердил дизассемблированием в 2017-м. Паттерн столь же актуален сейчас, как и тогда, на любой платформе, где такты дефицитны и каждая инструкция должна оправдать своё присутствие.

Итого

- Эффект сферы в Illusion отображает монохромное исходное изображение на вращающуюся сферу с помощью динамически сгенерированного Z80-кода.
- Таблицы подстановки кодируют геометрию сферы как расстояния пропуска пикселей. Рендерящий код генерируется из этих таблиц во время выполнения.
- Внутренний цикл использует `ADD A, A` и `ADD A, (HL)` для накопления пикселей в экранные байты, с переменным числом инструкций `INC L` для продвижения по исходным данным.
- Производительность: $101 + 32x$ тактов на выходной байт, где x зависит от позиции.
- Подход воплощает общий демосценовый паттерн: предвычислить геометрию, сгенерировать код, обращаться к памяти последовательно.

- Dark применил эти алгоритмы в Illusion (1996), а затем задокументировал их в Spectrum Expert (1997–98). Introspec провёл реверс-инжиниринг результата двадцать лет спустя, подтвердив техники.

Источники: Introspec, «Технический анализ Illusion от X-Trade» (Нуре, 2017); Dark, «Алгоритмы программирования» (Spectrum Expert #01, 1997). Ветка комментариев на Нуре включает вклады kotsoft, Raider и других.

Глава 7: Ротозумер и чанки-пиксели

«Трюк в том, что ты не вращаешь экран. Ты вращаешь свой проход по текстуре.» – перефразируя ключевую идею, стоящую за каждым когда-либо написанным ротозумером

Есть момент в Illusion, когда экран заполняется паттерном — текстурой, монохромной, повторяющейся — и затем она начинает вращаться. Вращение плавное и непрерывное, зум дышит туда-сюда, и всё это идёт в таком темпе, что забываешь: ты смотришь, как Z80 гонит пиксели на 3.5 МГц. Это не самый технически сложный эффект в демо. Сфера (Глава 6) сложнее математически. Точечный скроллер (Глава 10) жёстче по бюджету тактов. Но ротозумер — тот, что выглядит лёгким, а на Spectrum сделать что-то лёгким на вид — самый трудный трюк из всех.

Эта глава прослеживает две нити. Первая — анализ Introspec'a 2017 года ротозумера из Illusion от X-Trade. Вторая — статья sq 2022 года на Нуре об оптимизации чанки-пикселей, которая доводит подход до 4x4 пикселей и каталогизирует семейство стратегий рендеринга с точными подсчётами тактов. Вместе они отображают пространство решений: как работают чанки-пиксели, как их используют ротозумеры и какие компромиссы производительности определяют, работает ли твой эффект на 4 кадрах за экран или на 12.

Что на самом деле делает ротозумер

Ротозумер отображает 2D-текстуру, повернутую на некоторый угол и масштабированную на некоторый коэффициент. Наивный подход: для каждого экранного пикселя вычислить соответствующую текстурную координату через тригонометрическое вращение:

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

При 256x192 это 49 152 пикселя, каждый требующий двух умножений. Даже с 54-тактным умножением через таблицу квадратов (Глава 4) получается более пяти миллионов тактов — примерно 70 кадров процессорного времени. Эффект математически тривиален и вычислительно невозможен.

Ключевая идея в том, что преобразование *линейно*. Перемещение на один пиксель вправо по экрану всегда добавляет одинаковое (dx, dy) к текстурным

координатам. Перемещение на один пиксель вниз всегда добавляет одинаковое (dx' , dy'). Стоимость на пиксель коллапсирует от двух умножений до двух сложений:

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

Начинаем каждую строку с правильной текстурной координаты и шагаем на (dx , dy) для каждого пикселя. Внутренний цикл становится: прочитать текстель, продвинуть на (dx , dy), повторить. Два сложения на пиксель, никаких умножений. Подготовка к кадру — четыре умножения для вычисления векторов шага из текущего угла и масштаба. Всё остальное следует из линейности.

Это фундаментальная оптимизация, стоящая за каждым ротозумером на любой платформе. На Amiga, на PC, на Spectrum.

Пошаговое перемещение с фиксированной точкой на Z80

На 16-битной или 32-битной платформе dx и dy были бы значениями с фиксированной точкой: целая часть выбирает текстель, а дробная накапливает субпиксельную точность. На Z80 у нас нет ни регистров, ни пропускной способности для настоящих внутренних циклов с фиксированной точкой. Классическое решение для Spectrum — свести шаг к целочисленным приращениям — всегда ровно +1, -1 или 0 по каждой оси — и управлять соотношением шагов между осями для аппроксимации угла.

Рассмотрим поворот на 30 градусов. Точный вектор шага составит $(\cos 30, -\sin 30) = (0.866, -0.5)$. На машине с арифметикой с фиксированной точкой ты бы прибавлял 0.866 к координате столбца и вычитал 0.5 из координаты строки на каждый пиксель. На Z80 внутренний цикл вместо этого чередует два целочисленных шага: для одних пикселей шаг (+1 столбец, 0 строк), для других — (+1 столбец, -1 строка). Если распределить их примерно в соотношении 2:1 — два шага только по столбцу на каждый диагональный шаг — среднее направление аппроксимирует соотношение 0.866:0.5 обхода под углом 30 градусов. Это алгоритм линии Брезенхэма, применённый к обходу текстуры.

Коэффициент масштабирования определяет, сколько текстелей ты пропускаешь на каждый экранный пиксель. При масштабе 1.0 каждый текстель соответствует одному экранному пикセルю. При масштабе 2.0 ты пропускаешь каждый второй текстель, фактически приближая изображение. На Spectrum это контролируется удвоением инструкций обхода: вместо одного INC L на пиксель ты выполняешь два, шагая на 2 текстеля и получая 2-кратное увеличение. Промежуточные уровни масштабирования снова используют распределение в стиле Брезенхэма: одни пиксели шагают на 1, другие на 2, а соотношение управляет аккумулятором ошибки.

Покадровая стоимость вычисления этих параметров пренебрежимо мала: четыре обращения к таблице синусов, несколько умножений (или обращений к таблицам подстановки, см. Главу 4) и подготовительный проход Брезенхэма. Вся тяжёлая работа — во внутреннем цикле, который сведён к одним лишь инкрементам регистров и чтениям памяти.

Чанки-пиксели: размен разрешения на скорость

Даже при двух сложениях на пиксель запись 6 144 байт в чересстрочную видеопамять Spectrum за кадр непрактична — не если ты ещё хочешь обновить угол и оставить время на музыку. Чанки-пиксели решают это, снижая эффективное разрешение. Вместо одного текстеля на экранный пиксель ты отображаешь один текстель на блок 2x2, 4x4 или 8x8.

Illusion использует чанки-пиксели 2x2: эффективное разрешение 128x96, четырёхкратное сокращение работы. Эффект выглядит блочно вблизи, но при скорости, с которой текстура проносится по экрану, движение скрывает грубость. Глаз прощает низкое разрешение, когда всё движется.

Почему 2x2 — золотая середина

Выбор размера блока — это компромисс между тремя параметрами: визуальным качеством, скоростью рендеринга и памятью. При 2x2 ты получаешь 128x96 эффективных пикселей — достаточно, чтобы читать текст и распознавать узоры в текстуре. При 4x4 сетка 64x48 заметно грубее; мелкие детали текстуры становятся нечитаемыми, но эффект всё ещё «читается» как связная вращающаяся поверхность. При 8x8 остаётся лишь 32x24 блока, что соответствует разрешению сетки атрибутов — любые детали текстуры теряются, и эффект выглядит как цветные прямоугольники. Последний вариант может быть полезен для чисто цветовых эффектов (атрибутные туннели, Глава 9), но для пиксельного ротозумера практический диапазон — 2x2 или 4x4.

Расход памяти тоже имеет значение. Каждый чанки-пиксель занимает один байт, поэтому ротозумер 2x2 при разрешении 128x96 обрабатывает 12 288 текстелей за кадр. При ширине текстурной строки 256 байт (естественная ширина для 8-битного заворачивания) сама текстура занимает 256 байт на строку, умноженные на нужное количество строк. Версия 4x4 обрабатывает всего 3 072 текстеля, то есть внутренний цикл выполняет в четыре раза меньше итераций — но визуальная цена существенна.

На практике спектрумовские демо используют 2x2 для ключевых эффектов ротозумера, а 4x4 оставляют для ситуаций, когда ротозумер делит экран с другими эффектами (оверлеи бампмаппинга, композиции с разделённым экраном).

Трюк с кодировкой \$03

Кодировка рассчитана на внутренний цикл. Каждый чанки-пиксель хранится как \$03 (вкл.) или \$00 (выкл.). Это значение выбрано не случайно — оно кодирует ровно два установленных младших бита: %00000011. Посмотри, что происходит, когда четыре пикселя накапливаются в регистре A:

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)

Если все четыре пикселя «включены», результат — \$FF, все биты установлены. Если все четыре «выключены» (\$00), сдвиги и сложения дают \$00. Смешанные паттерны дают правильную полосу по 2 бита на пиксель: например, вкл-выкл-вкл-выкл даёт %11001100 = \$CC. Каждая пара битов в выходном байте соответствует одному чанки-пикселю. Поскольку каждый чанки-пиксель имеет ширину 2 экранных пикселя (2x2), 8-битный выходной байт покрывает ровно 8 экранных пикселей: четыре чанки-столбца по два пикселя каждый.

Ключевое свойство: поскольку мы всегда прибавляем только \$03 или \$00, переноса между полями пикселей не возникает. Двухбитные группы никогда не переполняются друг в друга. Именно это делает кодировку безветвленной — не нужны маски, не нужны операции OR, только ADD A,A и ADD A,(HL).

Внутренний цикл из Illusion

Дизассемблирование Introspec'a выявляет ядро рендерящей последовательности. HL проходит по текстуре; H отслеживает одну ось, L — другую:

```
; 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
```

Последовательность повторяется для третьего и четвёртого пикселей. inc l и dec h вместе прочерчивают диагональный путь по текстуре — а диагональный означает повёрнутый. Конкретная комбинация инструкций инкремента и декремента определяет угол вращения.

Шаг	Инструкции	Такты
Чтение пикселя 1	ld a,(hl)	7
Проход	inc l : dec h	8
Сдвиг + чтение пикселя 2	add a,a : add a,a : add a,(hl)	15
Проход	inc l : dec h	8
Сдвиг + чтение пикселя 3	add a,a : add a,a : add a,(hl)	15
Проход	inc l : dec h	8
Сдвиг + чтение пикселя 4	add a,a : add a,a : add a,(hl)	15
Проход	inc l : dec h	8
Вывод + продвижение	ld (de),a : inc e	~11
Итого на байт		~95

Introspec измерил примерно 95 тактов на 4 чанки.

Критическое наблюдение: направление прохода жёстко закодировано в потоке инструкций. Другой угол вращения требует других инструкций. Восемь основных направлений возможны с использованием комбинаций inc l, dec l, inc h, dec h и пор. Это означает, что рендерящий код меняется каждый кадр.

Самомодифицирующийся код на уровне байта

«Покадровая генерация кода» звучит экзотично, но механизм прост. Каждая инструкция обхода — это один байт в памяти. INC L — опкод \$2C. DEC L — \$2D. INC H — \$24. DEC H — \$25. NOP — \$00. Чтобы изменить направление обхода с «вправо и вверх» (INC L + DEC H) на «чисто вправо» (INC L + NOP), ты записываешь \$00 в байт, где сейчас находится \$25. Это и есть весь шаг генерации кода: LD A,\$00 : LD (walk_target),A. Несколько записей в поток инструкций — и внутренний цикл теперь обходит текстуру в другом направлении.

Адреса-цели известны на этапе ассемблирования. Каждое место SMC помечено меткой (например, .smc_walk_h_0:) и код подмены использует эти метки как буквальные адреса. Нет динамического выделения памяти, нет разбора инструкций, нет дизассемблирования в рантайме. Ты записываешь известные опкоды по известным адресам. У Z80 нет кэша инструкций, который нужно инвалидировать, нет конвейера, который нужно сбрасывать. Запись вступает в силу немедленно при следующем чтении с этого адреса.

В полностью развёрнутом внутреннем цикле (который Illusion использует для своих 16-байтных строк) пришлось бы подменить 64 места с инструкциями обхода: 4 пары инструкций обхода на каждый выходной байт, умноженные на 16 байт на строку. Подмена 64 байт стоит примерно $64 \times 13 = 832$ такта (T-state) (каждая LD (nn),A — 13 тактов (T-state)), что ничтожно мало по сравнению со 100 000+ тактами (T-state) рендеринга. Генератор кода дёшев. Важен генерированный код.

Покадровая генерация кода

Рендерящий код генерируется заново каждый кадр, с инструкциями прохода, подставленными под текущий угол:

Диапазон угла	Шаг H	Шаг L	Направление
~0 градусов	nop	inc l	Чисто вправо
~45 градусов	dec h	inc l	Вправо и вверх
~90 градусов	dec h	nop	Чисто вверх
~135 градусов	dec h	dec l	Влево и вверх
~180 градусов	nop	dec l	Чисто влево
~225 градусов	inc h	dec l	Влево и вниз
~270 градусов	inc h	nop	Чисто вниз
~315 градусов	inc h	inc l	Вправо и вниз

Для промежуточных углов генератор распределяет шаги неравномерно, используя накопление ошибки в стиле Брезенхэма. Вращение на 30 градусов чередует inc l : nop и inc l : dec h примерно в соотношении 2:1, аппроксимируя тангенс 30 градусов (1.73:1). Результирующий код — развёрнутый цикл, где каждая итерация имеет свою специфическую пару прохода, настроенную на текущий угол.

Стоимость рендеринга для 128x96 при чанки 2x2. Область 128x96 — это 96 строк пикселей, но каждый тексель 2x2 покрывает две строки пикселей, давая

48 текстурных строк. Каждая текстурная строка порождает 16 выходных байт (128 пикселей / 8 бит на байт, по 4 чанки-пикселя в каждом байте):

```
1,520 x 48 texel rows = 72,960 T-states total
```

Примерно 1 кадр на Pentagon (71 680 тактов (T-state) на кадр). Но это только голый внутренний цикл. Полный подсчёт включает:

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)
<hr/>		
Inner loop:	72,960 T	
Total per frame:	~ 95,460 T (= 1.33 Pentagon frames)	

На стандартном 48K/128K Spectrum при 69 888 тактах (T-state) на кадр рендеринг занимает примерно 1,4 кадра. Оценка Introspec в 4-6 кадров на экран учитывает более сложный путь кода в Illusion (который обрабатывает полный экран 256x192, а не только полосу 128x96) и стоимость музыкального движка, работающего в прерывании. На Pentagon с его чуть более длинным кадром (71 680 тактов (T-state)) и без спорной памяти внутренний цикл работает примерно на 3% быстрее.

Спорная память на 48K/128K Spectrum добавляет ещё одну скрытую стоимость. Во время верхних 192 строк развёртки ULA крадёт такты у процессора при обращении к нижним 16 КБ ОЗУ (\$4000-\$7FFF). Внутренний цикл читает из текстуры (которая должна быть выше \$8000, вне спорной памяти) и пишет в буфер (тоже выше \$8000), поэтому он полностью избегает конкуренции. Перенос буфера на экран, однако, пишет напрямую в видеопамять и будет замедлен спорной памятью, если пересечётся с периодом отображения. Поэтому демо синхронизируют перенос на экран с периодом бордюра или нижней частью отображения.

Перенос буфера на экран

Ротозумер рендерит во внеэкранный буфер, затем переносит в видеопамять. Черезстрочная раскладка экрана делает прямой рендеринг болезненным, а буферизация предотвращает разрывы.

Перенос использует стек:

```
pop hl ; 10T -- read 2 bytes from buffer
ld (screen_addr),hl ; 16T -- write 2 bytes to screen
```

Экранные адреса встроены как лiteralные операнды, предвычисленные с учётом чередования Spectrum — ещё один пример генерации кода. При 26 тактах на два байта полный перенос 1 536 байт стоит менее 20 000 тактов. Проход рендеринга — узкое место, не перенос.

Глубокое погружение: чанки-пиксели 4x4 (sq, Hype 2022)

Статья sq доводит чанки-пиксели до 4x4 — эффективное разрешение 64x48. Визуальный результат грубее, но выигрыш в производительности открывает двери для эффектов вроде бампмаппинга и чересстрочного рендеринга. Статья — образец методологии оптимизации: начать просто, итеративно улучшать, измерять на каждом шаге.

Подход 1: Базовый LD/INC (101 такт на пару). Загрузить чанки-значение, записать в буфер, продвинуть указатели. Узкое место — управление указателями: INC HL по 6 тактов набирается за тысячи итераций.

Подход 2: Вариант с LDI (104 такта — медленнее!). LDI копирует байт и автоинкрементирует оба указателя одной инструкцией. Но она также декрементирует BC, занимая регистровую пару. Накладные расходы на сохранение/восстановление делают его *медленнее* наивного подхода. Поучительный урок: на Z80 «умная» инструкция не всегда быстрая.

Подход 3: LDD двойной байт (80 тактов на пару). Организуя источник и назначение в обратном порядке, автодекремент LDD работает в твою пользу. Комбинированная двухбайтная последовательность использует это для 21% улучшения над базовым вариантом.

Подход 4: Самомодифицирующийся код (76-78 тактов на пару). Предгенирировать 256 процедур рендеринга, по одной на каждое возможное значение байта, с пиксельным значением, втытым как непосредственный операнд:

```
; 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
```

256 процедур занимают примерно 3 КБ. Рендеринг на пиксель падает до 76–78 тактов — на 23% быстрее базового, на 27% быстрее LDI.

Сравнение производительности

Подход	Такты/пару	Относительно	Память
Базовый LD/INC	101	1.00x	Минимум
Вариант LDI	104	0.97x	Минимум
LDD двойной байт	80	1.26x	Минимум
Самомодифицирующийся (256 проц.)	76–78	1.30x	~3 КБ

Самомодифицирующийся подход побеждает, но отрыв от LDD невелик. В 128К-демо 3 КБ легко найдутся. В 48К-продукции подход LDD может оказаться лучшим инженерным решением.

Исторические корни: Born Dead #05 и сценовая преемственность

sq отмечает, что эти техники развиваются из работы, опубликованной в Born Dead #05, российской демосценовой газете примерно 2001 года. Born Dead был одним из нескольких русскоязычных дисковых журналов, служивших техническими изданиями для демосцены ZX Spectrum. В отличие от западных PC-демосценовых публикаций, которые могли рассчитывать на оборудование уровня 486, спектрумовские журналы работали в условиях сообщества, всё ещё активно разрабатывавшего новые техники для машины 1982 года. Основополагающая статья описывала базовый чанки-рендеринг — идею о том, что битовый дисплей Spectrum можно трактовать как чанки-пиксельный буфер более низкого разрешения, выигрывая в скорости за счёт разрешения.

Вклад sq, двадцать один год спустя, состоял в систематической оптимизации и варианте с предгенерированными процедурами. Но между Born Dead #05 и статьёй sq 2022 года чанки-ротозумер появился в многочисленных демо для Spectrum. Illusion от X-Trade (ENLiGHT'96) была одной из первых полноценных реализаций. Среди других заметных примеров — GOA4K и Refresh от Exploder^XTM, работы 4D, а также более поздние произведения российской ипольской сцен. Техника распространялась отчасти через дизассемблирование — анализ Illusion, выполненный Introspec в 2017 году, сам по себе является примером сценовой традиции обучения через реверс-инжиниринг — и отчасти через неформальную сеть знаний: дисковые журналы, посты на BBS и прямое общение между кодерами.

Так эволюционирует сценовое знание: техника появляется в малоизвестном дисковом журнале, распространяется внутри сообщества, и двадцать один год спустя кто-то возвращается к ней с новыми замерами и новыми трюками. Цепочка от Born Dead к sq и к этой главе непрерывна.

Практика: построение простого ротозумера

Вот структура работающего ротозумера с чанки-пикселями 2x2 и шахматной текстурой.

Текстура. 256-байтная таблица, выровненная по странице, где каждый байт — \$03 или \$00, генерирующая полоски шириной 8 пикселей. Регистр H обеспечивает второе измерение; XOR H с индексом создаёт полноценную шахматную доску:

```

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

```

Таблица синусов и покадровая подготовка. 256-элементная таблица синусов, выровненная по странице, управляет вращением. Каждый кадр читает `sin(frame_counter)` и `cos(frame_counter)` (косинус через смещение индекса на 64) для вычисления векторов шага, затем подставляет в инструкции прохода внутреннего цикла правильные опкоды.

Цикл рендеринга. Внешний цикл задаёт начальную текстурную координату для каждой строки (шагая перпендикулярно направлению прохода). Внутренний цикл проходит по текстуре:

```
.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
```

Инструкции `inc l` — цели генератора кода. Перед каждым кадром они заменяются на подходящую комбинацию `inc l/dec l/inc h/dec h/nop` в зависимости от текущего угла. Для некардинальных углов аккумулятор ошибки Брезенхэма распределяет шаги по второстепенной оси вдоль строки, так что каждая инструкция прохода в развернутом цикле может отличаться от соседних.

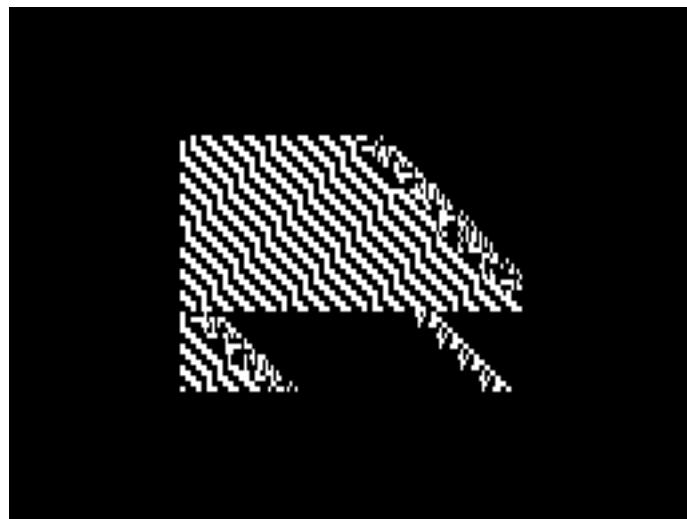


Рис. 17: Результат работы ротозумера — текстура вращается и масштабируется в реальном времени, отрендерена чанки-пикселями 2x2

Главный цикл. HALT для синхронизации, вычислить векторы шага, сгенерировать код прохода, отрендерить в буфер, перенести буфер на экран через стек, инкрементировать счётчик кадров, повторить.

Дизайн текстуры и обработка границ

Текстура — наиболее ограниченная структура данных в ротозумере. Каждое проектное решение внутреннего цикла — выравнивание по странице, поведение заворачивания, размеры степени двойки — восходит к тому, как текстура размещена в памяти.

Почему выравнивание по странице, почему 256 столбцов

Текстура выровнена по странице, чтобы H выбирал строку, а L — столбец. Это не просто удобно — это делает внутренний цикл возможным. INC L и DEC L автоматически заворачиваются на границе 256-байтной страницы — когда L переполняется с \$FF в \$00, H остаётся неизменным. Текстура заворачивается по горизонтали бесплатно, без каких-либо накладных расходов на ветвление. Если бы текстура не была выровнена по странице, инкременты L вызывали бы перенос в H, разрушая адрес строки. Пришлось бы использовать явное маскирование (AND \$3F после каждого шага), что добавило бы 4-8 тактов (T-state) на пиксель и уничтожило бы плотный внутренний цикл.

Вертикальная ось (H) тоже заворачивается, но в пределах всего диапазона строк, выделенных текстуре. Если ты выделишь 64 строки (страницы), H будет варьироваться от базовой страницы текстуры до base+63. INC H и DEC H спокойно выйдут за пределы текстуры в какую бы то ни было следующую память. Illusion решает это маскированием H по высоте текстуры в начале каждой строки (не на каждый пиксель — попиксельное маскирование было бы слишком дорогим). Это работает, потому что в пределах одной 16-байтной строки координата H изменяется максимум на 16 шагов, и если текстура достаточно высока относительно ширины строки, переполнение внутри строки не может достичь памяти, которая даст визуальный мусор. Текстура из 64 строк с 16 шагами H на строку имеет комфортный запас.

Выбор размера текстуры

Текстура должна иметь ширину, равную степени двойки (всегда 256, поскольку L — 8-битный), и в идеале высоту, равную степени двойки, для удобного маскирования. Типичные варианты:

- **256x256** (64 КБ): заполняет всю верхнюю память на 128K Spectrum. Максимальное разрешение, но не остаётся места для кода или буферов.
- **256x64** (16 КБ): практичный выбор. Помещается в один банк памяти 16 КБ на 128K-оборудовании. 6-битная маска высоты (AND \$3F) быстра и тайлится бесшовно.
- **256x32** (8 КБ): помещается на 48K Spectrum с местом для всего остального. Текстура повторяется заметнее, но для шахматного или полосатого узора повторение *и есть* дизайн.
- **256x16** (4 КБ): минимум. Подходит для очень простых паттернов вроде одноосевых полос.

Для неповторяющихся текстур (изображения, логотипы) высота должна быть не менее эффективной высоты экрана, делённой на коэффициент масштаби-

рования. Ротозумер 2x2 с 96 эффективными строками требует как минимум 96 текстурных строк, чтобы избежать видимого тайлинга при масштабе 1:1. При более высоких уровнях масштабирования строк нужно меньше, потому что камера «ближе» к поверхности текстуры.

А как насчёт границ экрана?

Экран Spectrum — 256x192, что составляет 32 байта в ширину на 192 строки. Если твой ротозумер заполняет полосу 128x96 в центре, ты никогда не приближаешься к краю видеопамяти. Но полноэкранный ротозумер при 256x192 (или даже 128x192 с чанки 2x2) должен обрабатывать случай, когда выходной адрес достигает области атрибутов по адресу \$5800. Простейший подход: рендерить в буфер и копировать только ту часть, которая помещается. Более агрессивный подход: обрезать количество строк до видимой области на этапе генерации кода, что позволяет избежать бесполезных вычислений, но усложняет цикл по строкам.

На практике большинство спектрумовских ротозумеров рендерят полосу меньше полного экрана. Визуальное обрамление — бордюр, заголовок, титры с музыкой — скрывает обрезку и высвобождает такты для других эффектов.

Пространство решений

Размер чанки-пикселя — самое важное проектное решение в ротозумере:

Параметр	2x2 (Illusion)	4x4 (sq)	8x8 (атрибуты)
Разрешение	128x96	64x48	32x24
Текселей/кадр	12 288	3 072	768
Стоимость внутр. цикла	~73 000 Т	~29 000 Т	~7 300 Т
Кадров/экран	~1,3	~0,5	~0,1
Визуальное качество	Хорошее в движении	Чанки, но быстро	Очень блочно
Применение	Ключевые эффекты	Бампмап-пинг, оверлей	Атрибутные эффекты

Версия 4x4 укладывается в один кадр с запасом для музыкального движка и других эффектов. Версия 2x2 занимает примерно 1,3-1,5 кадра (с учётом накладных расходов), но выглядит существенно лучше. Случай 8x8 — это атрибутный туннель из Главы 9.

Как только у тебя есть быстрый чанки-рендерер, ротозумер — лишь одно применение. Тот же движок приводит в действие **бампмаппинг** (читать разницу высот вместо сырых текстелей, выводить затенение), **чересстрочные эффекты** (рендерить чётные/нечётные строки в чередующихся кадрах, удваивая эффективную частоту кадров ценой мерцания) и **искажение текстуры** (менять направление прохода построчно для волнистых или рябящих эфектов).

Ротозумер 4x4 может делить кадр со скроллтекстом, музыкальным движком и переносом экрана. Работа sq была мотивирована именно этой универсальностью.

Три подхода к вращению текстуры

Всё вышеизложенное рассматривает ротозумер как одну технику с настраиваемым размером блока. Но «ротозумер» на Spectrum — это на самом деле семейство трёх различных подходов, каждый со своим внутренним циклом, своим визуальным характером и своим профилем производительности. Они разделяют одну и ту же математическую основу — линейные векторы шага, распределение углов в стиле Брезенхэма — но полностью расходятся на уровне рендеринга.

Вариант 1: Монокромный битмап (полное пикельное разрешение)

Чистейшая форма: каждый экранный пиксель соответствует одному текселю. Текстура монокромная — один бит на пиксель — поэтому чтение текселя означает проверку одного бита, а запись на экран означает установку или сброс одного бита. Никакой чанки-кодировки, никакой группировки блоков. Результат — повёрнутая текстура в полном разрешении 256x192 дисплея Spectrum.

Скелет внутреннего цикла выглядит примерно так:

```
; 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
```

Обрати внимание, что SET и RES работают только с (HL), (IX+d) или (IY+d) — не с (DE) и не с (BC). Это вынуждает HL служить указателем на экран, а DE — обрабатывать координаты текстуры.

Стоимость на пиксель жестокая: минимум 35-45 тактов (T-state) с ветвлением на каждом пикселе. На 49 152 пикселя это 1,5-2 миллиона тактов (T-state) только на проход рендеринга — примерно 21-28 кадров на стандартном Spectrum. Полнокраинный монокромный ротозумер при 50 кадрах в секунду — такого не будет.

Но никто не говорил, что нужно заполнять весь экран. Техника раскрывается при применении к меньшей области — полосе 128x64, круглому вышпорту,

маскированной зоне — или когда ты готов принять более низкую частоту кадров ради визуального эффекта вращения в полном разрешении. Она также прекрасно работает для эффектов искажения, где «вращение» неравномерно: варьирование векторов шага по строкам развёртки создаёт волновые искажения, эффекты бочки и вид «звуковой ряби», встречающийся в частях Illusion от Dark/X-Trade. Координатное отображение перестаёт быть простым вращением и превращается в построчную деформацию текстуры. Математика та же — пошаговое перемещение с фиксированной точкой вдоль направления — но само направление меняется на каждой строке.

Визуальная отдача поразительна. Там, где чанки-ротозумер 2x2 выглядит как вращающаяся мозаика, монохромная битмап-версия выглядит как вращающееся *изображение*. На машине, где каждый эффект борется за один и тот же бюджет кадра в 69 888 тактов (T-state), выделение нескольких кадров на рендеринг в полном разрешении — это осознанный эстетический выбор.

Вариант 2: Чанки-ротозумер (блоки 2x2 или 4x4)

Это техника, описанная в основной части этой главы. Каждый экранный блок (2x2 или 4x4 пикселя) соответствует одному текселю. Кодировка \$03/\$00, накопление через `add a,a : add a,(hl)`, подмена инструкций обхода — всё это нацелено на этот подход.

При 2x2 (эффективное разрешение 128x96) внутренний цикл работает приблизительно на 95 тактов (T-state) за выходной байт, создавая плавный, узнаваемый ротозумер, видимый в Illusion. При 4x4 (64x48) вариант sq с предгенерированными процедурами полностью устраняет накладные расходы цикла, снижая стоимость до 76–78 тактов (T-state) на выходную пару и оставляя место для мультиэффектных композиций в пределах одного кадра.

Чанки-ротозумер занимает золотую середину: достаточно быстрый для реального времени, достаточно детальный, чтобы нести на себе ключевой эффект. Это рабочая лошадка репертуара ротозумеров на Spectrum.

Вариант 3: Атрибутный ротозумер («пиксели» из блоков 8x8)

Область атрибутов Spectrum по адресам \$5800-\$5AFF хранит цветовую информацию для каждой знакомой ячейки 8x8 пикселей: 32 столбца на 24 строки, всего 768 байт. Каждый байт кодирует INK, PAPER, BRIGHT и FLASH для одного блока 8x8. Атрибутный ротозумер полностью игнорирует битмап и трактует эти 768 ячеек атрибутов как поверхность отображения. Каждая ячейка становится одним «пикселием» в изображении 32x24.

Внутренний цикл структурно идентичен чанки-версии — обходим координаты текстуры, читаем значение, записываем на выход — но выход — это область атрибутов, а значение «текселя» — это байт цветового атрибута, а не битовый паттерн. Эффективное разрешение — всего 32x24, а значит, весь проход рендеринга — это 768 итераций цикла обхода.

Расчёт:

~10 T-states per cell (read texel + write attribute + step)
 $768 \times 10 = \sim 7,680$ T-states total

Это примерно 11% одного кадра. Ты мог бы запустить атрибутный ротозумер девять раз подряд — и всё ещё оставался бы запас для музыкального движка. Стоимость настолько мала, что эффект по сути бесплатен.

Но визуальная отдача отличается от битмап-вариантов. Ты вращаешь не пиксели — ты вращаешь цветные блоки. При 32x24 мелкие детали невидимы. Вместо этого ты получаешь раскатывающееся поле цвета, яркую мозаику, которая вращается и дышит. Атрибутный ротозумер в Illusion использует именно это: ярко окрашенную текстуру (не монохромный битмап), отображённую через сетку атрибутов, создавая характерный вид «витража» из вращающихся цветовых полей, которым Illusion знаменита. Поля PAPER и INK в каждом байте атрибута дают два цвета на ячейку, поэтому тщательно спроектированная текстура может нести больше визуальной информации, чем подразумевает сырое разрешение.

Атрибутный ротозумер идеально подходит для фонов, переходов или как базовый слой с наложенными поверх пиксельными эффектами. Поскольку он пишет только в область атрибутов, битмап можно одновременно использовать для другого эффекта — скроллера, логотипа, поля частиц — работающего в своём ритме. Этот многослойный подход — визитная карточка мультиэффектных экранов демо на Spectrum.

Сравнение

Вариант	Эффективное разрешение	Байт записано/кадр	~Тактов (рендеринг)	Цвет	Типичное применение
Монохромный битмап	256x192 (или подобласть)	6 144 (полный экран)	1 500 000-2 000 000	1 бит	Главный эффект, искажение, деформация
Чанки 2x2	128x96	1 536	~73 000	1 бит	Ключевой ротозумер
Чанки 4x4	64x48	384	~29 000	1 бит	Мульти-эффект, оверлей
Атрибутный	32x24	768	~7 700	INK+PAPER, (2 цвета/ячейка-заливка, переход)	переход

Движение сверху вниз по таблице — это плавный обмен: разрешение на скорость, детализация на запас тактов. Монохромный битмап даёт всё, что может показать дисплей Spectrum, ценой, требующей полной отдачи. Атрибутная версия почти ничего не даёт в разрешении, но работает так быстро, что ротозумер становится лишь одним из инструментов в мультиэффектной композиции, а не главным событием.

Все четыре строки этой таблицы используют один и тот же базовый алгоритм. Векторы шага вычисляются одинаково. Распределение по Брезенхэму работает одинаково. Разница лишь в том, куда ты пишешь и сколько итераций выполняешь. Построив один ротозумер, ты построил их все.

Ротозумер в контексте

Ротозумер — не алгоритм вращения. Это *паттерн обхода памяти*. Ты проходишь по буферу по прямой линии, и направление прохода определяет, что ты видишь. Вращение — один из вариантов направления. Зум — выбор размера шага. Z80 не знает тригонометрии. Он знает INC L и DEC H. Всё остальное — интерпретация программиста.

В Illusion ротозумер стоит рядом со сферой и точечным скроллером. Все три разделяют одну архитектуру: предвычисленные параметры, сгенерированные внутренние циклы, последовательный доступ к памяти. Сфера использует таблицы пропусков и переменное количество INC L. Ротозумер использует направленно подставленные инструкции прохода. Точечный скроллер использует стековые таблицы адресов. Три эффекта, одна философия движка.

Dark построил все три. Introspec трассировал все три. Паттерн, связывающий их — урок Части II: вычислить необходимое до начала внутреннего цикла, сгенерировать код, который только читает-сдвигает-пишет, и поддерживать последовательный доступ к памяти.

Итого

- Ротозумер отображает повёрнутую и масштабированную текстуру, проходя по ней под углом. Линейность снижает стоимость на пиксель от двух умножений до двух сложений.
- Чанки-пиксели (2x2, 4x4) снижают эффективное разрешение и стоимость рендеринга пропорционально. Illusion использует 2x2 при 128x96; система sq использует 4x4 при 64x48.
- Внутренний цикл Illusion: ld a,(hl) : add a,a : add a,a : add a,(hl) с инструкциями прохода между чтениями. Стоимость: ~95 тактов на байт для 4 чанки-пикселей.
- Направление прохода меняется каждый кадр, требуя генерации кода — рендерящий цикл подставляется перед каждым кадром.
- Путь оптимизации sq для 4x4: базовый LD/INC (101 такт) к LDI (104 такта, медленнее) к LDD (80 тактов) к самомодифицируемому коду с 256 предгенерированными процедурами (76–78 тактов, ~3 КБ). Основан на более ранней работе в Born Dead #05 (~2001).
- Перенос буфера на экран через pop hl : ld (nn),hl по ~26 тактов на два байта.
- Ротозумер разделяет архитектуру со сферой (Глава 6) и точечным скроллером (Глава 10): предвычисленные параметры, сгенерированные внутренние циклы, последовательный доступ к памяти.

Источники: Introspec, «Технический анализ Illusion от X-Trade» (Hype, 2017); sq, «Chunky Effects on ZX Spectrum» (Hype, 2022); Born Dead #05 (~2001, оригинальные техники чанки-пикселей).

Глава 8: Мультиколор — Преодоление атрибутной сетки

«Мультиколор будет побеждён.» — DenisGrachev, Нуре, 2019

Каждый кодер ZX Spectrum знает правило. Два цвета на ячейку 8x8. Ink и paper. Вот что даёт ULA, и это всё, что она даёт. Если у твоего персонажа красная шапка и синее пальто и два цвета попадают в одну атрибутную ячейку, один из них проигрывает. Результат — кричащая окантовка, ошибочно окрашенные спрайты, персонажи, меняющие цвет при движении мимо декораций — это конфликт атрибутов, и он является определяющим визуальным ограничением платформы.

Конфликт атрибутов настолько фундаментален для идентичности Spectrum, что многие кодеры просто принимают его. Они проектируют вокруг него. Подбирают палитры для его минимизации. Ограничивают размеры спрайтов или избегают определённых цветовых сочетаний. Тридцать лет сетка 8x8 была фактом жизни.

Но ULA не знает этого.

ULA читает байты атрибутов по мере отрисовки экрана, строка развёртки за строкой. Она не читает все 768 байт атрибутов сразу. Она читает каждую строку из 32 атрибутов именно тогда, когда они ей нужны, восемь строк развёртки спустя читает ту же строку снова для следующей пиксельной линии внутри символьной строки, и так далее. Атрибут для любой ячейки читается восемь раз за кадр — по одному разу на каждую пиксельную строку в этой ячейке.

Трюк очевиден, как только его увидишь: если изменить байт атрибута между чтениями, ULA применит другой цвет к разным пиксельным строкам внутри одной ячейки. Вместо двух цветов на все восемь строк получаешь два цвета на группу строк. Сетка 8x8 ломается не потому, что оборудование перепроектировали. Она ломается потому, что ты перезаписал данные быстрее, чем оборудование могло их потребить.

Это мультиколор. Он известен как минимум с начала 2000-х, когда российский ZX-журнал Black Crow опубликовал алгоритм и пример кода в пятом выпуске. Но годами мультиколор оставался диковинкой — впечатляющей в демо, непрактичной в играх, потому что процессор тратил столько тактов на смену атрибутов, что ничего не оставалось на игровую логику.

А потом DenisGrachev придумал, как делать с этим игры.

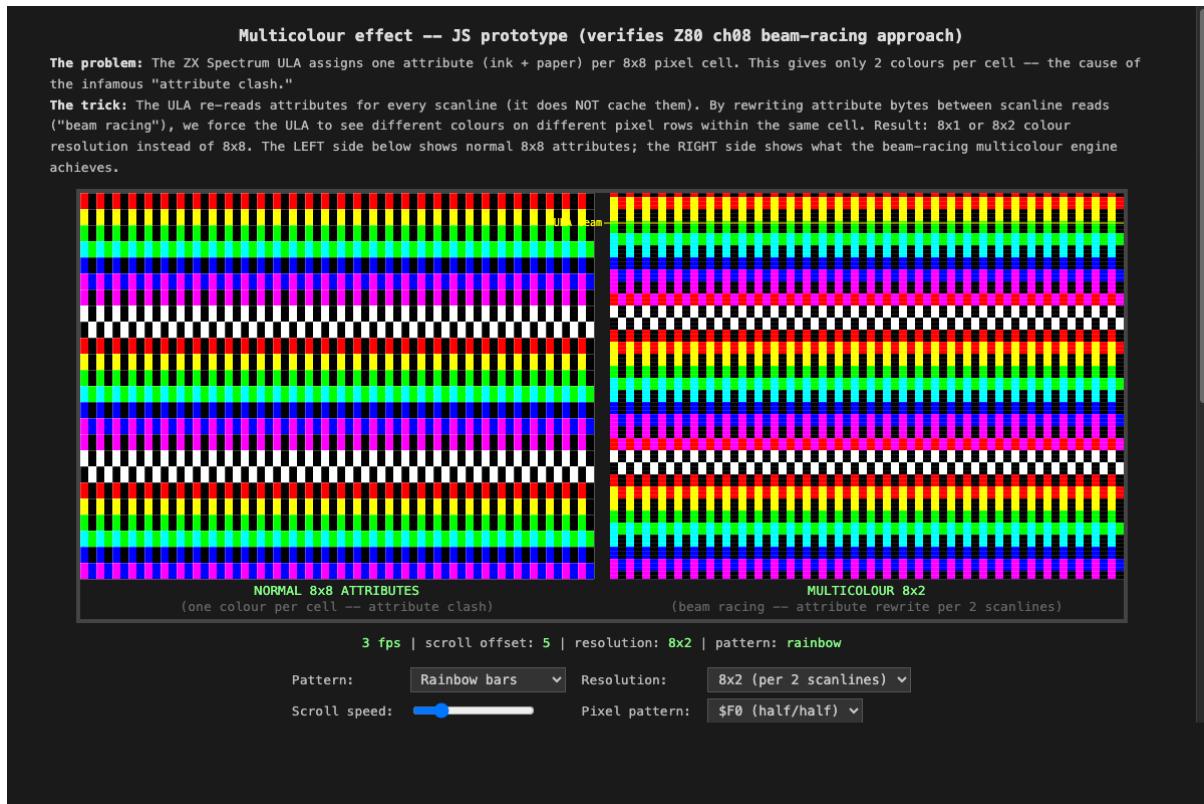


Рис. 18: Обычные атрибуты 8x8 vs мультиколорная гонка с лучом 8x2 — правая панель показывает, как перезапись атрибутов между строками развёртки увеличивает цветовое разрешение

Точка зрения ULA

Чтобы понять мультиколор, нужно увидеть экран с точки зрения ULA.

ULA рисует 192 видимые строки развёртки за кадр, сверху вниз. Каждая строка занимает 224 такта (T-state) (на Pentagon). Для каждой строки ULA читает 32 пиксельных байта и 32 байта атрибутов из памяти. Пиксельные байты определяют, какие точки — ink, а какие — paper. Байты атрибутов определяют, какими цветами на самом деле являются «ink» и «paper».

Внутри символьной строки (8 пиксельных линий) ULA читает одни и те же 32 байта атрибутов для каждой строки развёртки. Она не кэширует их — читает заново каждый раз. Это означает окно возможности между чтением атрибутов одной строки и следующей для изменения данных атрибутов.

«Традиционный» подход к мультиколору использует это напрямую. После HALT (который синхронизирует процессор с кадровым прерыванием) ты считаешь такты, чтобы знать точно, когда ULA прочитает каждую строку атрибутов. Затем, в промежутке между чтениями, перезаписываешь байты атрибутов новыми значениями. Когда ULA прочитает их для следующей строки, она увидит новые цвета.

Ограничение жёсткое: точный подсчёт тактов, изменение 32 байт между строками, затем ожидание следующей возможности. Процессор проводит почти всё время на этой бухгалтерии. В типичном традиционном мультиколорном движке можно менять атрибуты каждые 2 или 4 строки, давая цветовое разрешение 8x2 или 8x4. Но бюджет тактов, потреблённый кодом смены атрибутов, не оставляет почти ничего для игровой логики, рендеринга спрайтов или звука.

Поэтому мультиколор оставался в демо. Демо могут позволить себе тратить 100% процессора на визуальные эффекты. Игры — нет.

Прозрение LDPUSH

В январе 2019 года DenisGrachev опубликовал статью на Нуре под названием «Мультиколор будет побеждён» (Mul'tikolor budet pobezhdon). Заголовок был заявлением о намерениях. Он разрабатывал Old Tower, игру для ZX Spectrum с мультиколором 8x2 — атрибуты меняются каждые две пиксельные строки — и хотел объяснить, как он решил проблему бюджета тактов, делавшую мультиколор непрактичным в играх.

Ключевая идея из тех, что кажутся неизбежными задним числом: код, выводящий пиксельные данные, сам является дисплейным буфером.

Традиционный мультиколор разделяет код и данные. Где-то в памяти есть буфер байтов атрибутов и рендерящий код, копирующий их на экран в нужный момент. Техника DenisGrachev'a сливает их воедино. «Буфер» — это последовательность Z80-инструкций — а именно, LD DE, nn, за которым следует PUSH DE — и исполнение этих инструкций записывает дисплейные данные непосредственно в экранную память через указатель стека.

Вот как это работает.

LD DE,nn / PUSH DE

Инструкция LD DE,nn загружает 16-битное непосредственное значение в регистровую пару DE. Она занимает 10 тактов и имеет длину 3 байта: байт опкода \$11, за ним два байта данных (загружаемое значение, младший байт первым). Инструкция PUSH DE уменьшает SP на 1, записывает старший байт DE, снова уменьшает SP на 1, затем записывает младший байт. Результат: SP оказывается на 2 ниже, старший байт — по старшему адресу, младший байт — по младшему адресу. Она занимает 11 тактов.

Вместе LD DE,nn : PUSH DE стоят 21 такт, занимают 4 байта и записывают 2 байта данных на экран. «Данные» — это непосредственный операнд инструкции LD. Чтобы изменить то, что рисуется, ты не перезаписываешь дисплейный буфер — ты патчишь байты операнда внутри самих инструкций LD.

```
; 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
```

Строка развёртки пиксельных данных — 32 байта в ширину. Но нельзя заполнить все 32 байта одним PUSH, потому что PUSH пишет вниз (SP декрементируется), а данные должны появляться слева направо на экране. Раскладка экранной памяти Spectrum справляется с этим: в пределах одной строки развёртки последовательные байты находятся по возрастающим адресам. PUSH пишет по убывающим адресам. Поэтому данные выходят наоборот — последний записанный байт оказывается по наименьшему адресу, что соответствует самому левому байту на экране.

Это значит, что LDPUSH-последовательность строится в обратном порядке отображения. Первая LD DE,nn : PUSH DE в коде записывает два самых правых байта строки. Последняя записывает два самых левых. При исполнении push'ы заполняют строку справа налево.

Сколько помещается в строку развёртки?

DenisGrachev не заполняет все 32 байта. Движок GLUF (который приводит в действие Old Tower и другие игры) использует игровую область шириной 24 символа, обрамлённую бордюрами с каждой стороны. Это 24 байта на строку развёртки в игровой области.

При 4 байтах кода на 2 байта вывода нужно 48 байт кода для заполнения 24 байт экрана. Но есть дополнительный байт на начальную LD SP,nn и смену атрибутов между группами строк. DenisGrachev сообщает о 51 байте на строку генерированного кода.

Красота подхода: нет отдельного прохода рендеринга. Инструкции, заполняющие экран, ЯВЛЯЮТСЯ исполнимым кодом. Когда нужно обновить тайл или спрайт, ты патчишь непосредственные операнды инструкций LD. Когда дисплейный код выполняется, он выводит пропатченные данные. Код — это данные. Данные — это код.

Указатель стека как курсор

При исполнении SP указывает на правый край текущей строки развёртки в экранной памяти. Каждый PUSH записывает 2 байта и уменьшает SP на 2, перемещая «курсор записи» влево по строке. В конце вывода одной строки SP указывает на левый край. Затем код корректирует SP на правый край следующей строки и повторяет.

Фундаментальное ограничение: прерывания должны быть отключены, пока SP перехвачен. Если бы прерывание сработало, процессор записал бы адрес возврата в экранную память, повредив изображение. Это значит, что код мультиколорного рендеринга выполняется с DI и повторно включает прерывания через EI только после восстановления SP в настоящий стек. Весь проход рендеринга — все 192 видимые строки — происходит в одном непрерываемом блоке.

Движок GLUF: мультиколор в настоящей игре

Old Tower была доказательством концепции. GLUF (игровой фреймворк DenisGrachev'a) стал продакшн-движком. Вот цифры:

Параметр	Значение
Разрешение мультиколора	8x2 (атрибуты меняются каждые 2 пиксельные строки)
Игровая область	24x16 символов (192x128 пикселей)
Буферизация	Двойная (два набора дисплейного кода)
Размер спрайта	16x16 пикселей
Размер тайла	16x16 пикселей
Звук	25 Гц (каждый второй кадр)
Тактов на кадр	~70 000 (почти весь бюджет Pentagon)

Разрешение 8x2 означает, что движок меняет атрибуты четыре раза на символьную строку вместо одного. Каждая символьная строка содержит 8 строк развёртки; смена атрибутов каждые 2 строки даёт четыре различные цветовые полосы внутри одной символьной ячейки. Символ, обычно ограниченный ink-na-paper, внезапно получает до восьми цветов (два на полосу, четыре полосы). На практике эффект разителен — спрайты и тайлы показывают куда больше цветовых деталей, чем архитектура Spectrum была рассчитана обеспечить.

Двойная буферизация

GLUF поддерживает два полных набора LDPUSH-дисплейного кода. Пока один набор исполняется (рисуя текущий кадр), другой патчится новыми данными тайлов и спрайтов для следующего кадра. Это устраняет мерцание, которое возникло бы при модификации дисплейного кода во время его исполнения.

Цена — память. Каждый набор дисплейного кода покрывает игровую область 24x16 символов по 51 байту на строку, умноженных на 128 строк: примерно 6 500 байт на буфер. Два буфера потребляют около 13 000 байт. На 128K Spectrum с банковой памятью это управляемо, но значительно — требуется тщательное планирование памяти для остальных ресурсов игры.

Двухкадровая архитектура

Вот где инженерия становится жёсткой. GLUF не рендерит полный кадр каждую 1/50 секунды. Он использует двухкадровую архитектуру:

Кадр 1: Сменить атрибуты для мультиколорного эффекта, затем отрендерить столько тайлов, сколько возможно, в буфер дисплейного кода. Смена атрибутов — критичная по времени часть; она должна произойти в точно правильный момент относительно луча развёртки.

Кадр 2: Закончить рендеринг оставшихся тайлов, затем наложить спрайты на буфер дисплейного кода.

Разделение необходимо, потому что объём работы просто не влезает в один кадр. Рендеринг тайлов в LDPUSH-буфер означает патчинг байтов операндов внутри дисплейного кода — для каждого пикселя тайла ты вычисляешь, на какую инструкцию LD он влияет, и записываешь новое значение в правильное байтовое смещение. Это не простое блочное копирование. Чередующаяся структура дисплейного кода (опкод-данные-данные-опкод-данные-данные...) означает, что рендеринг тайлов включает разрозненные записи, а не последовательные.

Общий бюджет рендеринга — примерно 70 000 тактов на кадр — почти весь бюджет Pentagon в 71 680. Что остаётся, едва хватает на игровую логику, обработку ввода и периодическое обновление звука.

Звук работает на 25 Гц вместо 50. Каждый второй кадр движок полностью пропускает обновление звука, чтобы вернуть эти такты на рендеринг. Игрок не замечает половинную частоту обновления для простых звуковых эффектов. Для музыки частота 25 Гц означает, что каждая нота длится вдвое больше кадров, что требует написания музыкального движка специально под это ограничение.

Что видит игрок

Игрок не видит ничего из этого. Он видит скролл-игру с большим количеством цветов, чем должен иметь Spectrum. Спрайты перемещаются поверх декораций без обычного конфликта атрибутов. Тайлы показывают затенение, текстуру и многоцветные детали, невозможные со стандартными атрибутами 8x8. Игра ощущается так, будто принадлежит более мощной платформе.

Именно это убедительно демонстрирует работа DenisGrachev'a: мультиколор — не демо-трюк. Это техника игрового движка. Инженерия экстремальна — двойные буфера, двухкадровый рендеринг, звук на 25 Гц — но результат — играбельная игра с визуалом, действительно ломающим воспринимаемые пределы Spectrum.

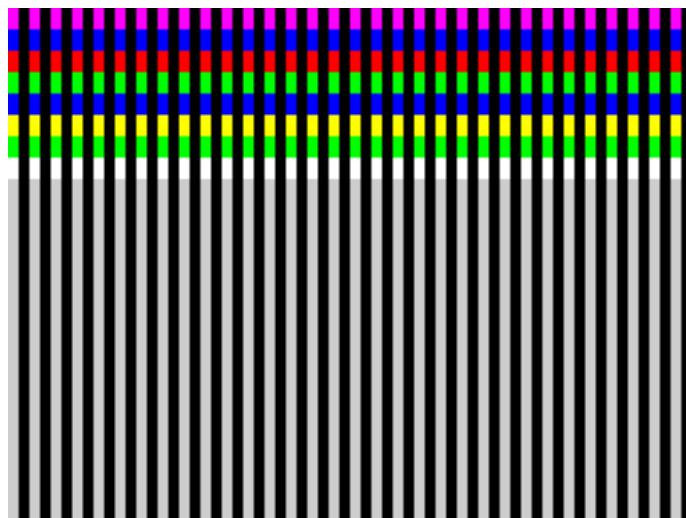


Рис. 19: Мультиколорный эффект — разрешение атрибутов 8x2 даёт каждой знакоместной ячейке до восьми цветов вместо двух

Ringo: мультиколор другого рода

В декабре 2022 года DenisGrachev опубликовал вторую статью на Hype: «Ringo Render 64x48». Если GLUF расширил атрибутную сетку до 8x2, то Ringo вообще отбросил её и построил нечто более близкое к чанки-пиксельному фреймбуферу с поточечным цветом.

Подход концептуально прост и технически изощрён.

Паттерн 11110000b

Заполни каждый пиксельный байт экранной памяти значением \$F0 — двоичное 11110000. Левые четыре пикселя каждого байта установлены (цвет ink), правые четыре — сброшены (цвет paper). Теперь, если изменить атрибут этой ячейки, левая половина отобразит цвет ink, а правая — цвет paper. Одна ячейка шириной 8 пикселей показывает два различных цвета бок о бок.

Со стандартными атрибутами 8x8 это даёт сетку 64x24 независимо окрашенных «пикселей», каждый шириной 4 реальных пикселя и высотой 8. Неплохо, но не революционно.

Переключение двух экранов

Трюк, делающий Ringo работающим: ZX Spectrum 128K имеет два экранных буфера. Экран 0 находится по адресу \$4000, Экран 1 — по адресу \$C000 (в банке 7). Один OUT в порт \$7FFD переключает, какой экран отображает ULA.

Ringo подготавливает оба экрана с паттерном 11110000b, но с *разными* атрибутами на каждом экране. Затем он переключается между двумя экранами каждые 4 строки развёртки. Эффект: внутри каждой символьной строки высотой 8 пикселей верхние 4 строки показывают атрибуты Экрана 0, а нижние 4 — атрибуты Экрана 1. Каждая половина может задавать независимые цвета ink и paper.

В сочетании с пиксельным паттерном 11110000b это даёт:

- 2 цветовых столбца на символьную ячейку (левые 4 пикселя = ink, правые 4 = paper)
- 2 цветовые строки на символьную ячейку (верхние 4 строки развёртки от Экрана 0, нижние 4 от Экрана 1)
- Итого: 4 независимо окрашенные субъячейки на ячейку 8x8

По всему экрану: 64 столбца x 48 строк = **3 072 независимо окрашенных пикселя**, каждый размером 4x4 реальных пикселя. Эффективное разрешение 64x48 с полным поточечным цветом из 15-цветной палитры Spectrum.

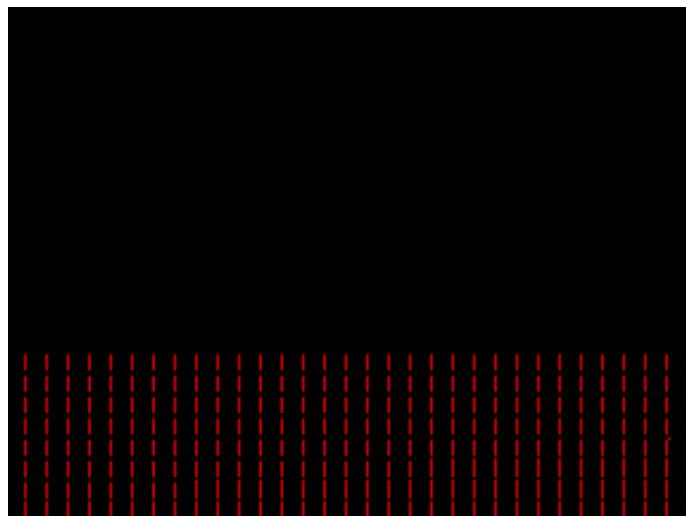


Рис. 20: Двухэкранный мультиколор на 128K Spectrum — переключение двух экранов по методу Ringo даёт сетку 64x48 с независимым цветом для каждого пикселя

Это принципиально другой подход по сравнению с мультиколором GLUF 8x2. GLUF меняет атрибуты синхронно с лучом, требуя точного тайминга и потребляя огромные бюджеты тактов. Ringo использует аппаратное переключение двух экранов, требующее лишь одной инструкции OUT каждые 4 строки развёртки. Накладные расходы процессора на само переключение экранов минимальны.

Куда уходят такты

Дешёвое переключение экранов означает больше тактов для игровой логики. Но рендеринг в два экрана одновременно не бесплатен. Каждое обновление тайла и спрайта должно быть записано и в Экран 0, и в Экран 1, потому что игрок видит композицию обоих.

Спрайты Ringo — 12x10 «пикселей» в сетке 64x48, что означает 12 байт атрибутов в ширину и 10 атрибутных строк в высоту (разделённых между двумя экранами). Каждый спрайт занимает 120 байт данных. Рендеринг спрайтов использует макросы с фиксированным числом тактов — последовательности инструкций с известным, постоянным временем исполнения, критичные для поддержания синхронизации с переключением экранов.

Рендеринг тайлов сложнее. DenisGrachev предгенерирует код рендеринга тайлов в страницы памяти, используя паттерны `pop af : or (hl)`:

```
; 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

```

`pop af` — стековый трюк: данные тайла организованы как таблица в стековом формате в памяти. SP указывает на данные тайла, и POP читает два байта за раз. `or (hl)` комбинирует цвет тайла с тем, что уже есть на экране, позволяя прозрачные тайлы и слоёные фоны.

Горизонтальный скроллинг

Ringo реализует горизонтальный скроллинг со смещением в полсимвола. Поскольку каждый «пиксель» в сетке 64x48 имеет ширину 4 реальных пикселя, скроллинг на один «пиксель» означает сдвиг паттерна `11110000b` на 4 бита. Но паттерн фиксирован — его нельзя легко сдвинуть, не нарушив цветовой трюк.

Вместо этого DenisGrachev скроллит, перемещая данные атрибутов. Скролл на один пиксель сдвигает все атрибуты на один столбец влево или вправо и рисует новый столбец на краю. Поскольку меняются только атрибуты (пиксельный паттерн остаётся фиксированным `$F0`), скролл — это просто блочное копирование байтов атрибутов. Для сетки 64x48 это 48 байт на смещение столбца (один байт на строку), куда дешевле пиксельного скроллинга.

Для суб-«пиксельного» скроллинга — плавного движения в пределах 4-пиксельного столбца — DenisGrachev чередует `$F0` и `$E0` (или аналогичные сдвинутые паттерны) в пиксельных данных. Это требует больше бухгалтерии, но достигает смещения в полсимвола, создавая иллюзию горизонтального разрешения в 128 столбцов.

Традиционный мультиколор: подход на прерываниях

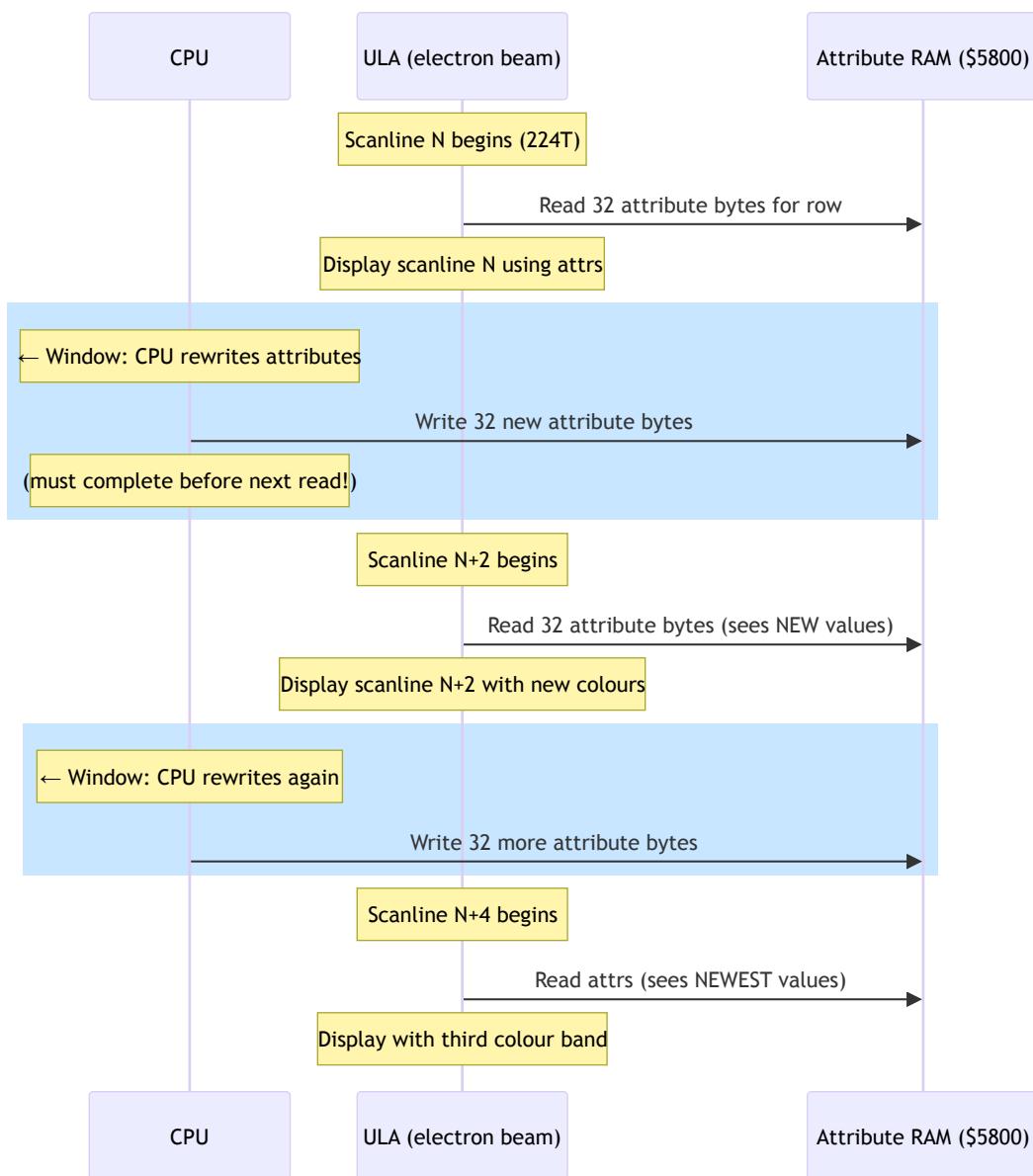
Прежде чем перейти к практике, стоит разобраться в «классическом» подходе, который GLUF и Ringo вытеснили. Традиционный мультиколор концептуально прост: менять атрибуты в точно нужный момент, и ULA отобразит разные цвета на разных строках развёртки.

Техника работает так:

1. Выполнить HALT для синхронизации с кадровым прерыванием. После HALT процессор находится в известной позиции по тактам относительно начала отображения.
2. Считать такты от HALT. ULA читает каждую строку из 32 атрибутов в известный момент каждой строки развёртки. Дополняя NOP'ами или другими инструкциями известной длины, ты можешь расположить свой код в точно нужный момент.
3. В точный такт, когда ULA закончила читать атрибуты текущей строки (но до того, как прочитает их для следующей), перезаписать 32 байта атрибутов новыми значениями.

4. Подождать, пока ULA прочитает новые значения, затем перезаписать снова для следующей смены.

Тайминг жёсток. Каждая строка развёртки занимает 224 такта на Pentagon. ULA читает 32 байта атрибутов в начале каждой строки, и процессор должен изменить все 32 байта в промежутке до следующего чтения. С $LD (HL), A : INC L$ по 11 тактов на байт запись 32 байт занимает 352 такта — больше одной целой строки развёртки. Нельзя менять каждую строку. В лучшем случае можно менять каждую вторую (разрешение 8x2), если использовать самый быстрый метод вывода (на основе PUSH), и даже тогда допуски по таймингу бритвенно тонки.



Гонка: При 224 тактах на строку развёртки запись 32 байтов через $LD (HL), A : INC L$ стоит 352Т — больше одной строки развёртки. Именно поэтому разрешение 8x2 (смена каждые 2 строки развёртки) — практический предел при программной перезаписи атрибутов, и именно поэтому LDPUSH объединяет вывод пикселей и атрибутов,

чтобы избежать отдельного прохода по атрибутам.

Практический результат: традиционный мультиколор потребляет 80–90% процессора на управление атрибутами. В демо, где мультиколор *и есть* эффект, это приемлемо. В игре — летально. На игровую логику, определение коллизий или звук тактов не остаётся.

Техника LDPUSH DenisGrachev'a решает это, объединяя вывод атрибутов с выводом пикселей. Один и тот же код, записывающий пиксельные данные, записывает и атрибуты, и оба встроены в исполнимые инструкции. Нет отдельной фазы «управления атрибутами», пожирающей бюджет. Проход рендеринга обрабатывает всё.

Врезка: Black Crow #05 — ранний мультиколор

Техника смены атрибутов между строками развёртки была задокументирована уже в 2001 году в Black Crow #05, российском журнале ZX Spectrum-сцены, распространявшемся на TRD-образах дисков. Статья представляла базовый алгоритм — синхронизация с растром, подсчёт тактов, смена атрибутов — вместе с рабочим примером кода.

Black Crow важен как исторический маркер. К 2001 году демосцена ZX Spectrum эволюционировала далеко за пределы коммерческого срока жизни платформы, и кодеры систематически каталогизировали трюки, выводящие оборудование за пределы его проектных спецификаций. Мультиколор был одной из многих техник, распространявшихся через экосистему сценовых журналов: Spectrum Expert, ZX Format, Born Dead, Black Crow и позже онлайн-платформа Hype.

Вклад DenisGrachev'a, почти два десятилетия спустя, был не в изобретении мультиколора, а в решении его практических проблем для разработки игр. Журнальные статьи задокументировали, что *возможно*. DenisGrachev показал, что *применимо*.

Палитра Spectrum и мультиколор

Краткая заметка о механике цвета, потому что визуальное воздействие мультиколора целиком зависит от палитры.

ZX Spectrum имеет 15 цветов: 8 базовых (чёрный, синий, красный, пурпурный, зелёный, голубой, жёлтый, белый) и 7 ярких вариантов (яркий чёрный совпадает с чёрным, поэтому только 7 дополнительных). Каждый байт атрибута задаёт цвет ink (3 бита), цвет paper (3 бита), флаг BRIGHT (1 бит, применяется и к ink, и к paper одновременно) и флаг FLASH (1 бит).

С мультиколором 8x2 каждая символьная ячейка получает четыре атрибутные строки вместо одной. Каждая строка задаёт независимые цвета ink и paper. Это до восьми цветов на ячейку (два на строку, четыре строки) — хотя на практике ограничение BRIGHT (применяется к обоим ink и paper) сужает реальные комбинации.

С подходом Ringo 64x48 каждая субъячейка полностью независима. 15-цветная палитра доступна в каждой из 3 072 позиций. Результат ближе к тому, что 8-битные домашние компьютеры с более мощным оборудованием — MSX2, Amstrad CPC — могли достичь аппаратно. На Spectrum это достигается полностью программно, эксплуатируя временное соотношение между процессором и ULA.

Практика: мультиколорный игровой экран

Построим упрощённый мультиколорный рендерер. Цель: игровая область шириной 24 символа с цветовым разрешением 8x2 и одним движущимся спрайтом поверх. Это не дотянет до полного набора функций GLUF, но продемонстрирует ядро техники LDPUSH и двухкадрового подхода к рендерингу.

Шаг 1: Буфер дисплейного кода

Нужен блок памяти, заполненный парами LDPUSH-инструкций. Для каждой строки развёртки в 24-символьной игровой области нужно 12 пар LD DE,nn : PUSH DE (каждая пара выводит 2 байта, 12 пар выводят 24 байта = полная ширина игровой области).

```
; 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)
```

Полный дисплейный код для 128 строк развёртки (16 символьных строк x 8 строк развёртки каждая) при примерно 51 байте на строку — около 6 500 байт.

Шаг 2: Смена атрибутов внутри дисплейного кода

Каждые две строки развёртки дисплейный код должен включать запись атрибутов. Между LDPUSH-последовательностями для строк N и N+2 вставляем код, перезаписывающий 32 байта атрибутов текущей символьной строки:

```
; 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

```

Смена атрибутов встроена непосредственно в поток дисплейного кода. Она выполняется в точно нужный момент, потому что *позиционирована* в точно нужном месте последовательности инструкций. Никакого подсчёта тактов. Никакого NOP-заполнения. Структура кода гарантирует тайминг.

Шаг 3: Рендеринг тайлов через патчинг операндов

Чтобы нарисовать тайл в игровой области, нужно пропатчить байты операндов инструкций LD в буфере дисплейного кода. Тайл 16x16 пикселей покрывает 2 байта в ширину и 16 строк развёртки в высоту. В дисплейном коде эти 2 байта — операнд конкретной инструкции LD DE. Чтобы обновить тайл:

```

; 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)

```

При 19 тактах на IX-индексированную запись это недёшево. Для полного тайла 16x16 (16 строк x 2 байта x 2 патча на байт): примерно 1 200 тактов на тайл. В 24x16-символьной игровой области с тайлами шириной 2 символа может быть до 192 тайлов. Даже с частичным обновлением (перерисовка только изменившихся тайлов) рендеринг тайлов доминирует в бюджете.

Вот почему GLUF разбивает рендеринг на два кадра. Кадр 1 обрабатывает критичную по времени смену атрибутов и рендерит столько тайлов, сколько возможно. Кадр 2 завершает тайлы и компонует спрайты поверх.

Шаг 4: Наложение спрайтов

Спрайты рендерятся поверх тайлов той же техникой патчинга операндов, но с дополнительным шагом: сохранить исходные байты операндов перед перезаписью, чтобы спрайт можно было стереть в следующем кадре, восстановив сохранённые данные.

```

; 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
```

Механизм сохранения/восстановления — мультиколорный эквивалент рендеринга спрайтов методом грязных прямоугольников. Сохраняешь то, что было, рисуешь спрайт, отображаешь, затем восстанавливашь сохранённые байты для стирания спрайта перед рисованием в новой позиции.

Шаг 5: Главный цикл

```
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
```

Этот каркас отражает основной ритм: два кадра на логический игровой кадр, дисплейный код исполняется с выключенными прерываниями, рендеринг тайлов и спрайтов происходит между проходами отображения. Чередование рендеринга и игровой логики в рамках двухкадровой структуры — инженерное сердце мультиколорного игрового движка.

Что значит работа DenisGrachev'a

Достижение DenisGrachev'a — не в изобретении мультиколора — техника была известна. Оно в решении инженерной проблемы: как уместить мультиколорный рендеринг, тайловые движки, наложение спрайтов, двойную буферизацию, звук и игровую логику в один бюджет кадра. Двухкадровая архитектура, объединённый буфер кода-как-данных и компромисс звука на 25 Гц — это решения игрового движка, а не демо-трюки. Ringo пошёл дальше, обменяв цветовое разрешение (64x48 против пиксельной сетки 192x128 у GLUF) на более дешёвый путь рендеринга через переключение двух экранов.

Итого

- **Конфликт атрибутов** (два цвета на ячейку 8x8) — определяющее визуальное ограничение Spectrum. ULA читает атрибуты построчно, а не покадрово — если менять их между чтениями, получаешь больше цветов на ячейку.
 - Техника **LDPUSH** сливает дисплейные данные с исполнимым кодом: последовательности LD DE,nn : PUSH DE записывают пиксельные данные в экранную память при исполнении, а непосредственные операнды служат дисплейным буфером. Патчинг операндов меняет то, что рисуется.
 - **GLUF** достигает мультиколора 8x2 в 24x16-символьной игровой области с двойной буферизацией дисплейного кода, 16x16 спрайтами и тайлами и двухкадровой архитектурой, разносящей рендеринг на 2/50 секунды.
 - **Ringo** использует пиксельный паттерн 11110000b с переключением двух экранов каждые 4 строки для достижения сетки 64x48 с поточечным цветом — принципиально другой компромисс, отдающий предпочтение цветовой независимости над пространственным разрешением.
 - **Традиционный мультиколор** (смена атрибутов на прерываниях) концептуально проще, но потребляет 80–90% процессора, делая его непрактичным для игр.
 - Журнал **Black Crows #05** задокументировал мультиколор уже в 2001 году. Вклад DenisGrachev'a — в том, что он сделал его практическим для разработки игр.
 - Работа DenisGrachev'a демонстрирует, что техники демосцены — это инженерные инструменты, а не только демо-трюки. Различие между «возможно в демо» и «применимо в игре» — это инженерная задача.
-

Попробуй сам

1. **Построй дисплейный буфер.** Напиши программу, генерирующую блок пар LD DE,nn : PUSH DE в памяти, направь SP на экранную память и исполни блок. Ты должен увидеть паттерн на экране, соответствующий выбранным непосредственным значениям. Измени значения и повторно исполни, чтобы увидеть обновление экрана.
2. **Добавь смену атрибутов.** Расширь дисплейный буфер, включив запись

атрибутов каждые 2 строки. Заполни чередующиеся полосы разными цветами. Ты должен увидеть горизонтальные цветные полосы внутри одной символьной строки — доказательство работы мультиколорного эффекта.

3. **Пропатчи тайл.** Напиши подпрограмму, которая берёт пиксельный тайл 16x16 и патчит его в дисплейный буфер в заданной позиции, модифицируя байты операндов LD. Нарисуй несколько тайлов для заполнения игровой области.
4. **Подвигай спрайт.** Реализуй сохранение/восстановление фонового патчинга: перед рисованием спрайта сохрани байты операндов, которые он перезапишет. После отображения кадра восстанови сохранённые байты. Двигай спрайт на один символ за кадр и убедись, что он перемещается чисто, без следов.
5. **Измерь бюджет.** Используй тайминговую обвязку с цветом бордюра из Главы 1, чтобы измерить, сколько кадра потребляет твой дисплейный код. На Pentagon красная полоса должна почти заполнить бордюр — GLUF использует ~70 000 из доступных 71 680 тактов. Посмотри, сколько остаётся на игровую логику.

Источники: DenisGrachev, «Мультиколор будет побеждён» (Hype, 2019); DenisGrachev, «Ringo Render 64x48» (Hype, 2022); Black Crow #05 (ZXArt, 2001). Игровые движки: Old Tower, GLUF, Ringo от DenisGrachev'a.

Глава 9: Атрибутные туннели и хаос-зумеры

«Это ОЧЕНЬ ГЛЮЧНОЕ демо. Это САМАЯ сложная вещь, которую я когда-либо делал в демо — без шуток.» – Introspec, file_id.diz партийной версии Eager (to live), ЗВМ Open Air 2015

Летом 2015 года Introspec сел делать то, за что раньше не брался. Демо, которое вышло — Eager (to live), выпущенное под лейблом Life on Mars на ЗВМ Open Air — заняло первое место в компо ZX Spectrum демо. Оно крутилось две минуты, зацикленное, и каждый кадр рендерился на 50 Гц с настоящими цифровыми барабанами, подмешанными в вывод AY-чипа. Визуальным центром был туннель, который, казалось, ввинчивался в экран, цвета расходились наружу органическими волнами. Многие зрители предположили, что задействована тяжёлая пиксельная манипуляция. На самом деле туннель ни разу не затронул ни одного пикселя. Весь эффект жил в области атрибутов.

Эта глава — making-of. Мы разберём два ключевых визуальных эффекта из Eager — атрибутный туннель и хаос-зумер — прослеживая творческое мышление наряду с кодом. По пути мы встретим скриптовый движок, намекающий на архитектуру синхронизации, рассмотренную в Главе 12, и философский спор о предвычислении, разделявший ZX-сцену годами. Но начнём там, где начал Introspec: глядя на 768 байт и осознавая, что их достаточно.

Атрибутная сетка как фреймбуфер

Каждый ZX Spectrum-кодер знает область атрибутов по адресам \$5800–\$5AFF. Каждый из 768 байт управляет цветами ink и paper для блока пикселей 8x8, образуя сетку 32x24. В разработке игр атрибуты — источник головной боли из-за конфликта атрибутов. В Главе 8 мы видели, как мультиколорные движки перезаписывают атрибуты синхронно с лучом развёртки для борьбы с сеткой 8x8. Атрибутный туннель делает обратное: он принимает сетку.

Идея обезоруживающе проста. Если заполнить пиксельную память фиксированным паттерном — скажем, чередующимися полосами ink/paper или шахматной доской — то байт атрибута сам по себе определяет, что зритель видит в каждой ячейке 8x8. Измени цвета ink и paper, и визуальное содержимое ячейки полностью меняется. Теперь у тебя 32x24 «пикселя» цвета, каждый — байт

атрибута. Запись полного кадра означает запись 768 байт. Никакого чередования адресов экрана. Никаких битовых манипуляций. Никакой попиксельной отрисовки. Просто линейное блочное копирование в RAM атрибутов.

При 32x24 разрешение ужасно по любым обычным стандартам. Но Introspec строил не обычный эффект. Он строил туннель.

Подумай, как туннель выглядит с точки зрения зрителя. «Устье» туннеля — центр экрана — туда притягивается взгляд. Стены уходят к краям. Вблизи центра детали мелки и размыты глубиной. У краёв стены близко, и видна текстура. Это красиво ложится на дисплей с переменным разрешением: грубое разрешение в центре (где туннель далеко и детали не важны) и более тонкое у краёв (где они важны).

Introspec пошёл дальше с псевдо-чанки-рендерингом. В центре экрана несколько атрибутных ячеек разделяют один цвет, создавая более крупные «пиксели». К краям каждая ячейка 8x8 получает своё значение. Глаз принимает блочный центр, потому что там устье туннеля — глубина естественно уничтожает детали. Периферийное зрение улавливает более тонкое разрешение на краях, создавая впечатление более высокой точности, чем данные реально содержат.

Вот первый урок Eager: атрибутная сетка — не ограничение, которое нужно обходить. Это фреймбуфер, с которым можно работать.

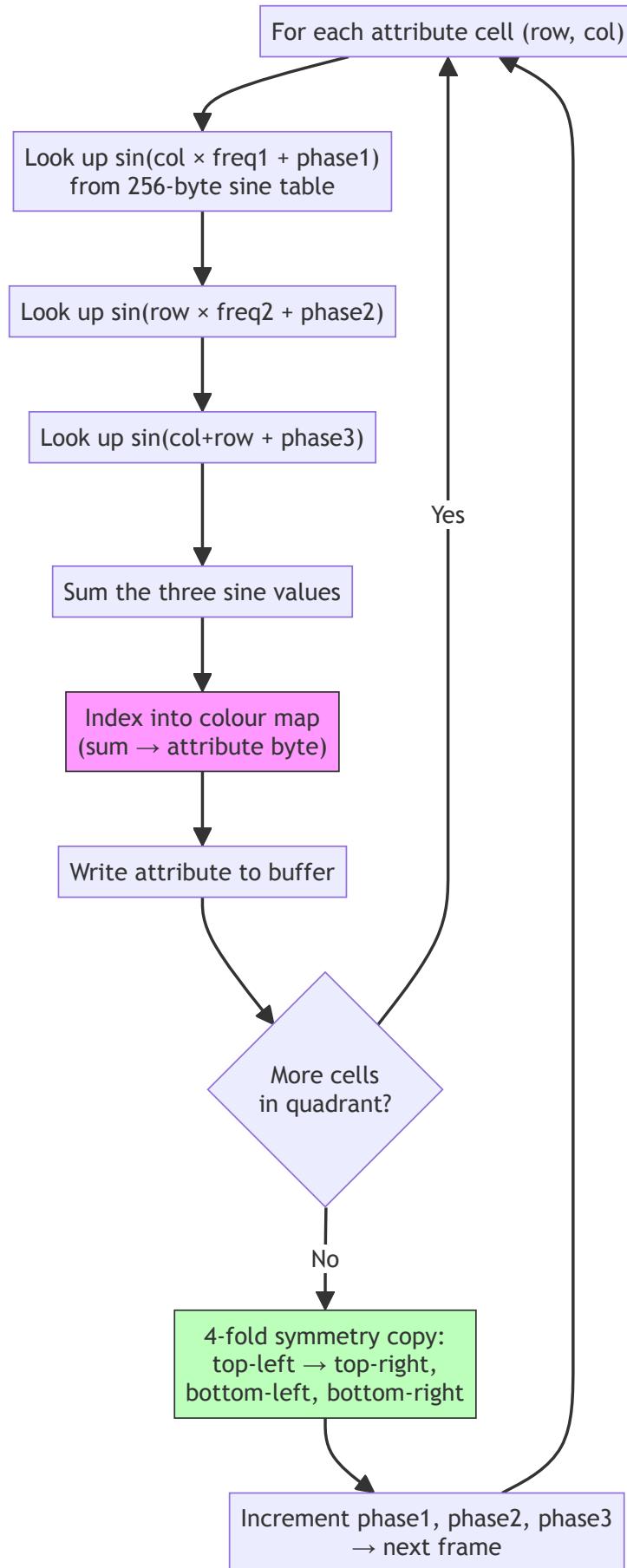
Плазма: цветовой движок

Цвета туннеля происходят не из сохранённой текстуры, наложенной на трубу. Они исходят из плазменного расчёта — классического подхода с суммой синусоид, ставшего демосценовым стандартом со времён Amiga, адаптированного здесь под атрибутную палитру Spectrum.

Основная идея: для каждой позиции в сетке 32x24 суммируем несколько синусоид с разными частотами и фазами. Результат, после приведения к доступному цветовому диапазону, определяет байт атрибута. Меняем фазы со временем — и плазма анимируется, создавая органическое, волнистое течение.

На Z80 это означает поиск по таблицам. 256-байтная таблица синусов, выровненная по странице для индексации одним регистром, обеспечивает базовую функцию. Для каждой ячейки ищем $\sin(x * \text{freq1} + \text{phase1}) + \sin(y * \text{freq2} + \text{phase2}) + \dots$, где умножения на частоту — на самом деле просто сложения индекса (умножить на 2 = искать каждую вторую запись, на 3 = сложить индекс с собой дважды). Накопленное значение индексирует цветовую карту, выдающую байт атрибута.

Форма туннеля неявна, не явна. Нет вычисления расстояния от центра, нет таблицы углов, нет преобразования в полярные координаты. Вместо этого параметры частот и фаз плазмы подобраны так, что результирующий цветовой паттерн естественно образует концентрические кольца на экране. Кольца возникают из интерференции синусоид, подобно тому как паттерны муара возникают из наложения сеток. Подстрой параметры — и кольца стянутся к центру, создавая иллюзию глубины — взгляда вниз в туннель.



Ключевое наблюдение: Здесь нет вычисления расстояния от центра, нет таблицы углов, нет преобразования в полярные координаты. Форма туннеля возникает из интерференции синусоид — концентрические кольца появляются естественным образом при наложении частот. Вычисляется только одна четверть (16×12); остальное отражается.

Это дешевле настоящего геометрического туннеля (который потребовал бы попиксельного поиска расстояния и угла) и даёт визуально богатый результат. Компромисс — меньшая геометрическая точность, но при разрешении 32×24 геометрическая точность никогда и не стояла на повестке.

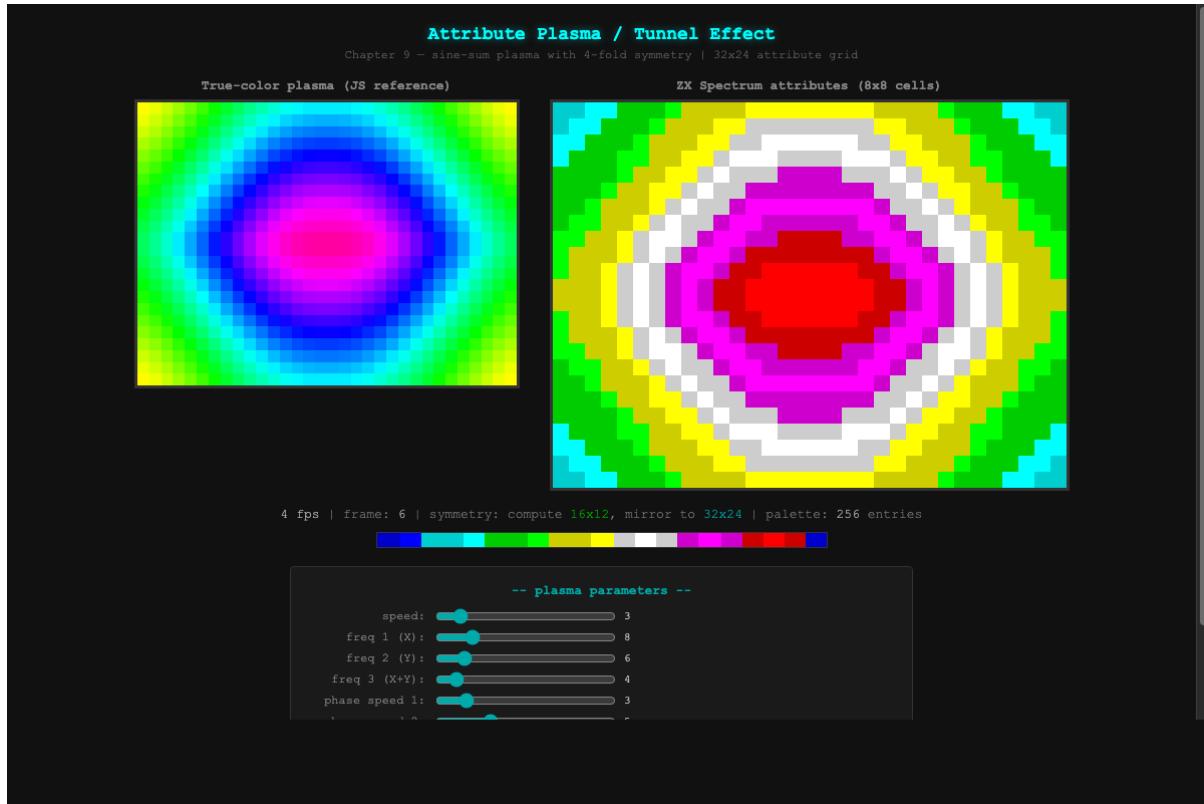


Рис. 21: Алгоритм плазмы — эталонная визуализация суммы синусов в true-colour (слева) vs вывод через 15-цветные атрибуты ZX Spectrum (справа), демонстрирующий, как четырёхсторонняя симметрия вдвое сокращает вычисления

Четвёртная симметрия: разделяй и властвуй

Даже при 32×24 расчёт плазмы для всех 768 ячеек каждый кадр дорог на Z80 с частотой 3.5 МГц. Introspec сократил нагрузку вчетверо классической оптимизацией: использовать естественную симметрию туннеля.

Туннель, видимый анфас, симметричен относительно горизонтальной и вертикальной осей. Если рассчитать одну четверть экрана — верхний левый блок 16×12 — можно скопировать её в три остальные четверти отражением. Верхний левый в верхний правый — горизонтальное отражение. Верхний левый в нижний левый — вертикальное. Верхний левый в нижний правый — оба.

Подпрограмма копирования компактна. В реализации Introspec'a HL указывает на исходный байт в верхней левой четверти, а три адреса назначения (верхний правый, нижний левый, нижний правый) поддерживаются комбинацией абсолютных адресов и регистровой пары BC:

```
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
```

Инструкции ld (nn),a и ld (mm),a используют абсолютную адресацию — целевые адреса встроены непосредственно в код, подставленные через самомодификацию или генерацию кода для каждой позиции ячейки. Инструкция ldi в конце выполняет двойную функцию: копирует байт из (HL) в (DE) для позиции верхней левой четверти в атрибутном буфере и автоинкрементирует оба HL и DE, одновременно декрементируя BC. Это значит, что счётчик цикла, продвижение указателя источника и одна из четырёх записей — всё свёрнуто в одну двухбайтную инструкцию.

Суммарная стоимость: менее 15 тактов на байт для четырёхстороннего копирования. Для 192 исходных байт (одна четверть 768-байтной области атрибутов) это примерно 2 880 тактов на заполнение всего экрана. При 3.5 МГц с бюджетом кадра ~70 000 тактов подавляющее большинство кадра остаётся на плазменный расчёт, музыкальный движок и воспроизведение цифровых барабанов, сделавшее Eager особенным.

Адреса (nn) и (mm) — литеральные двухбайтные значения, вшитые в инструкции LD (addr),A, подставленные через самомодификацию или генерацию кода для каждой позиции ячейки. Это стандартная демосценовая практика: отсутствие кэша инструкций у Z80 означает, что самомодифицирующийся код выполняется надёжно.

Хаос-зумер

Второй крупный визуальный эффект в Eager — хаос-зумер. Если туннель гладок и ограничен, зумер угловат и фрактоподобен — поле атрибутных данных, зумирующееся к зрителю или от него, с новыми деталями, приступающими на краях по мере продвижения зума.

«Хаос» — от визуального результата, а не от алгоритма. Эффект зумирует в область атрибутных данных, увеличивая центр, пока края смещаются внутрь. Поскольку исходные данные содержат паттерны на нескольких масштабах, зумирование выявляет самоподобную структуру, которую глаз воспринимает как фрактальную.

Реализация опирается на развёрнутые последовательности ld hl,nn : ldi. Каждый ld hl,nn загружает новый адрес источника — позицию в исходном буфере для выборки данной выходной ячейки. Следующий ldi копирует из (HL) в (DE), продвигая DE к следующей выходной позиции. Адреса источника организованы так, что ячейки вблизи центра экрана выбирают из соседних позиций

исходных данных (увеличение), а ячейки у краёв — из далеко разнесённых позиций (сжатие). Меняй маппинг со временем — и зум анимируется.

```
; 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)
```

Ключевая оптимизация: поскольку `ldi` автоинкрементирует `DE`, никогда не нужно вычислять или загружать адрес назначения. Вывод всегда записывается последовательно в RAM атрибутов. Варыируют только адреса источника, и они встроены непосредственно в поток инструкций как непосредственные операнды. Это делает зумер длинной последовательностью пар `ld hl,nn : ldi` — концептуально просто, но каждая пара занимает лишь 5 байт (3 для `ld hl,nn` + 2 для `ldi`) и 26 тактов. Для полной четвертиэкранной области из 192 ячеек это примерно 5 000 тактов чистого копирования, плюс четырёхстороннее копирование симметрии поверх.

Сложность в том, что адреса источника меняются каждый кадр по мере продвижения зума. Обновление 192 двухбайтных адресов, встроенных в код, стоило бы почти столько же, сколько само копирование. Тут в дело вступает генерация кода.

Генерация кода: Processing пишет Z80

Introspec не писал развёрнутый код зумера вручную. Последовательности адресов различны для каждого уровня зума, и вычислять их в рантайме сожрало бы бюджет кадра. Вместо этого он написал генератор кода на Processing, Java-основанной среде для креативного программирования. Скетч Processing вычислял для каждого кадра и каждой выходной ячейки, какую исходную ячейку выбирать, а затем выдавал полный .a80-файл, содержащий развёрнутую последовательность `ld hl,nn : ldi` со всеми заполненными адресами. sjasmplus компилировал этот сгенерированный исходник вместе с написанным вручную кодом движка.

Конвейер: Processing вычисляет маппинг зума, записывает .a80-исходник, ассемблер компилирует его, и в рантайме скриптовый движок выбирает, какой предгенерированный кадр исполнять. Z80 не вычисляет маппинг. Он лишь воспроизводит его.

Это обменивает память на скорость — предгенерированный код для всех кадров зума занимает значительный объём RAM, отсюда требование 128K — но стоимость в рантайме на кадр минимальна.

Вопрос запилятора

ZX-сцена давно и порой горячо относится к предвычислениям. Российская демосцена породила термин *запилятор* — примерно «предвычислятор» — для демо, которые сильно зависят от предгенерированных данных, а не от вычислений в реальном времени. Слово несёт лёгкий оттенок неодобрения. Если РС делает всю интересную работу, что на самом деле делает Spectrum? Это демо или слайд-шоу?

Ответ Introspec'а характерно нюансирует. Искусство, утверждает он, не в самом вычислении, а в *проектировании того, что предвычислять*. Выбор правильного маппинга зума, правильной интерполяции, правильного способа декомпозиции задачи так, чтобы предгенерированный код уместился в память, а воспроизведение шло на 50 Гц — это инженерия. Processing-скетч не пишет себя сам. Структура Z80-кода, делающая воспроизведение эффективным, не возникает автоматически. Творчество живёт в архитектуре, а не в том, содержит ли внутренний цикл `add` или `ldi`.

И он прав. Визуальное качество хаос-зумера зависит от исходных данных, функции маппинга, кривой зума, цветовой палитры и взаимодействия с музыкой. Всё это — художественные решения. Факт, что расчёт адресов происходит на этапе компиляции, а не в рантайме — деталь реализации, которая обеспечивает визуальное качество, невозможное при вычислениях в реальном времени на 3.5 МГц. Ограничения машины — её память, набор инструкций, тайминг — формировали каждое решение. То, что формирование отчасти происходило в Processing, а отчасти в Z80-ассемблере, не уменьшает результата.

Для целей этой книги практический вывод таков: генерация кода — законная и мощная техника. Если твоему эффекту нужны расчёты, превышающие бюджет кадра Z80, рассмотри перенос их на этап сборки. Макроязык твоего ассемблера, Lua-скрипт внутри sjasmplus или внешняя программа на Python или Processing — все они могут служить генераторами кода. Z80 получает возможность делать то, что он делает лучше всего: копировать данные на максимальной скорости.

Вкус скриптового движка

Eager содержит больше, чем туннель и зумер. Оно крутится две минуты, с множеством визуальных вариаций, переходов и воспроизведением цифровых барабанов, придающим демо ритмический пульс. Координация всего этого — скриптовый движок, тема, которую мы подробно исследуем в Главе 12 при обсуждении синхронизации с музыкой. Но краткий набросок здесь подготавливает к той дискуссии.

Движок Introspec'а использует два уровня скриптов. **Внешний скрипт** управляет последовательностью эффектов: проигрывать туннель N кадров, перейти к зумеру, вернуться к туннелю с другими параметрами, и так далее. **Внутренний скрипт** управляет вариациями внутри одного эффекта: сменой частот плазмы, ротацией цветовой палитры, изменением скорости зума.

Критически важная команда в скриптовом языке — то, что Introspec называет **kWORK**: «сгенерировать N кадров, затем показывать их независимо». Это ключ к асинхронной генерации кадров Eager. Движок предрендерит несколько

кадров текущего эффекта в буферы памяти. Затем, пока эти кадры отображаются (по одному за обновление экрана), движок может заниматься другой работой — например, воспроизводить сэмпл цифрового барабана через AY-чип.

Эта асинхронная архитектура и сделала Eager таким сложным в создании. Когда происходит удар барабана, процессор поглощён воспроизведением цифрового сэмпла. Генерация кадров останавливается. Визуальный эффект выживает на предрендеренных кадрах, пока барабан не закончится и генерация не возобновится. При частых ударах генератор отстает; между ударами догоняет. «Мой мозг плохо справляется с асинхронным кодированием», — написал Introspec в file_id.diz партийной версии. Честное изнemожение в этой записке отражает реальность переплетения критичной по таймингу аудиоподпрограммы с конвейером генерации кадров на машине с одним потоком и без операционной системы.

Мы вернёмся к этой архитектуре в Главе 12, где также рассмотрим технику цифровых барабанов n1k-o и двойную буферизацию атрибутных кадров, на которой строится вся система.

Making-of: хронология и вдохновение

Eager разрабатывался с июня по август 2015 года. Introspec говорил, что изначальное вдохновение пришло от увиденного эффекта twister в **Bomb** от Atebit — визуального трюка, использующего манипуляции атрибутами для создания иллюзии трёхмерного вращающегося столба. «Что ещё можно сделать только с атрибутами?» — вопрос, запустивший проект.

Музыку написал n1k-o (из Skrju), чей трек задал ритмическую структуру демо. Гибридная техника барабанов — цифровой сэмпл для атаки, AY-огибающая для затухания — была находкой n1k-o, и она определила всё архитектурное решение о построении движка асинхронной генерации кадров. Без барабанов Eager могло бы быть более простым демо. С ними оно стало тем, что Introspec назвал «самой сложной вещью, которую я делал в демо».

Разработка уложилась примерно в десять недель. Пати-версия, представленная на ZBM Open Air 2015, всё ещё содержала баги — file_id.diz нёс благодарность diver'у из 4D+TBK «за крутую подсказку» и извинения за нестабильность. Финальная версия исправила проблемы тайминга на разных моделях Spectrum (128K, +2, +2A/B, +3, Pentagon — все только на 3,5 МГц, без турбо). Это перекрёстное опыление — сценер из одной группы передаёт технический инсайт кодеру из другой — именно так эволюционирует ZX-демосцена.

Практика: построение упрощённого атрибутного туннеля

Построим упрощённую версию атрибутного туннеля Eager. Мы реализуем:

1. Фиксированный пиксельный паттерн в растровой памяти (чтобы атрибутам было что окрашивать).

2. Плазменный расчёт по атрибутной сетке 32x24.
 3. Четвёртную симметрию для сокращения расчёта до одной четверти.
 4. Анимацию через инкремент фазы плазмы каждый кадр.

Это не дотянет до визуальной изощрённости Eager — мы опускаем псевдо-чанки-пиксели переменного размера, генерируемый кодом зумер и асинхронную архитектуру. Но продемонстрирует ядро принципа: атрибутная сетка как фреймбуфер.

Шаг 1: Заполнение пиксельной памяти паттерном

Нужен фиксированный пиксельный паттерн, чтобы оба цвета ink и paper были видны. Простая шахматная доска подойдёт:

```
; 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
```

Каждая ячейка 8x8 теперь будет отображать чередующиеся пиксели ink и paper. При смене атрибута шахматная доска покажет оба цвета.

Шаг 2: Таблица синусов

Выравниваем 256-байтную таблицу синусов по странице для быстрой индексации. Её можно сгенерировать при ассемблировании через Lua-скрипting `siasmplus` или предвычислить и включить как бинарные данные:

```
ALIGN 256
sin_table:
    LUA ALLPASS
    for i = 0, 255 do
        -- Sine scaled to 0..63 (6-bit unsigned)
        sj.add_byte(math.floor(math.sin(i * math.pi / 128) * 31 + 32))
    end
ENDLUA
```

Шаг 3: Плазма для одной четверти

Рассчитываем плазменное значение для каждой ячейки верхней левой четверти 16x12. Результат — индекс в цветовую таблицу, выдающую байт атрибута:

```
; 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

```

Это намеренно упрощено. Продакшн-версия использовала бы самомодифицирующийся код для встраивания значений фаз (7 тактов на загрузку вместо 19 для IX-индексированных) и 256-байтную цветовую таблицу подстановки вместо цепочки rrca.

Шаг 4: Четырёхстороннее копирование

Упрощённый подход копирует строку за строкой, зеркально отражая горизонтально для правой половины и индексируя снизу для нижней. Но продакшн-

техника — однопроходное копирование Introspec'а, которое мы видели ранее, — записывает все четыре квадранта одновременно из одного исходного байта, используя самомодифицирующиеся инструкции `ld (nn)`, а с предпредставленными адресами. Практический код для этого паттерна — в каталоге примеров главы как `tunnel_4way.a80`.

Шаг 5: Главный цикл

```
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
```

Разные приращения фаз заставляют плазменные слагаемые вращаться с разными скоростями. Экспериментируй с этими значениями — даже малые изменения дают кардинально разные визуальные текстуры.

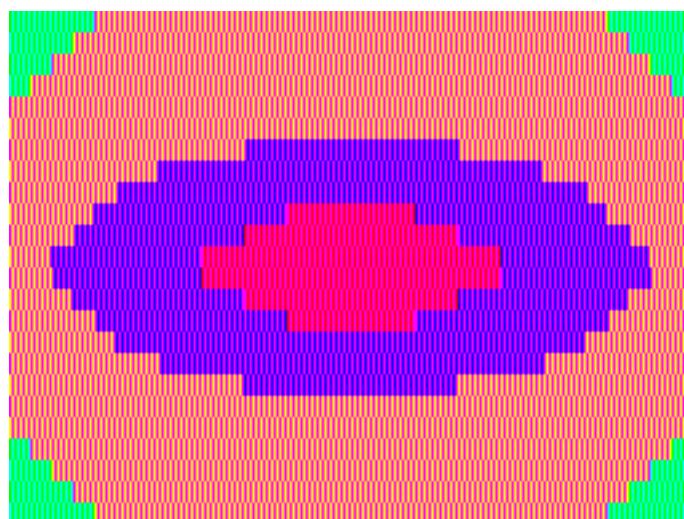


Рис. 22: Плазменный эффект на атрибутах — интерференция синусоид создаёт волнообразные цветовые узоры по всей сетке атрибутов 32x24

Ключевое наблюдение

Атрибутная сетка И ЕСТЬ твой фреймбуфер для этого эффекта. Ты никогда не трогаешь пиксельную память после начального заполнения шахматной

доской. Вся анимация состоит из записи 768 байт за кадр в \$5800–\$5AFF. Чередование экранных адресов Z80, делающее пиксельные манипуляции такими мучительными, совершенно неуместно. Область атрибутов линейна. Копирование быстро. Визуальный результат на 50 Гц плавен и на удивление убедителен.

Это урок, который Introspec извлёк из создания Eager, и он применим далеко за пределами туннелей. Всякий раз, когда цветовая система ZX Spectrum тебя расстраивает, подумай об инверсии задачи. Вместо борьбы с атрибутной сеткой используй её. Эти 768 байт — самый дешёвый полноэкранный анимационный буфер на этой машине.

Источники

- Introspec, «Making of Eager», Hype, 2015 (hype.retroscene.org/blog/demo/261.html)
- Introspec, file_id.diz из Eager (to live) партайная версия, ЗВМ Open Air 2015
- Introspec, «Код мёртв» (Code is Dead), Hype, 2015
- Introspec, «За дизайн» (For Design), Hype, 2015

Далее: Глава 10 переносит нас к точечному скроллеру Illusion и четырёхфазной цветовой анимации Eager — где два обычных кадра и два инвертированных создают иллюзию палитры, которой у Spectrum нет.

Глава 10: Точечный скроллер и 4-фазная цветовая анимация

«Два обычных кадра и два инвертированных кадра. Глаз видит среднее.» – Introspec, Making of Eager (2015)

ZX Spectrum отображает два цвета на ячейку 8x8. Текст прокручивается по экрану с той скоростью, которую может обеспечить процессор. Это фиксированные ограничения — аппаратура делает то, что делает, и никакая изобретательность не изменит кремний.

Но изобретательность может изменить то, что воспринимает зритель.

Эта глава объединяет две техники из двух разных демо, разделённых почти двадцатью годами, но связанных общим принципом. Точечный скроллер из *Illusion* от X-Trade (ENLiGHT'96) отрисовывает текст как подпрыгивающее облако отдельных точек, каждая из которых размещается ценой всего 36 тактов (T-states). 4-фазная цветовая анимация из *Eager* от Introspec (3BM Open Air 2015) чередует четыре тщательно подготовленных кадра на частоте 50 Гц, чтобы обмануть глаз и заставить его видеть цвета, которые оборудование не в состоянии воспроизвести. Одна техника эксплуатирует пространственное разрешение — размещая точки где угодно, без привязки к знакоместам. Другая эксплуатирует временное разрешение — циклически меняя кадры быстрее, чем глаз может уследить. Вместе они демонстрируют две главные оси обмана на ограниченном оборудовании: пространство и время.

Часть 1: Точечный скроллер

Что видит зритель

Представь сообщение — «ILLUSION BY X-TRADE» — отрисованное не сплошными блочными символами, а как поле отдельных точек, каждая точка — один пиксель. Текст плавно дрейфует по экрану горизонтально. Но точки не лежат на плоских строках развёртки. Они подпрыгивают. Всё точечное поле волнообразно колеблется по синусоиде, каждый столбец смешён по вертикали относительно соседних, создавая впечатление текста, рябящего на поверхности воды.

Шрифт как текстура

Шрифт хранится в памяти как битовая текстура — один бит на точку. Если бит равен 1, точка появляется на экране. Если бит равен 0, ничего не происходит. Ключевое слово здесь — *прозрачность*. В обычном рендере ты записываешь каждую пиксельную позицию. В точечном скроллере прозрачные пиксели практически бесплатны. Ты проверяешь бит, и если он нулевой — пропускаешь. Только установленные пиксели требуют записи в видеопамять.

Это означает, что стоимость отрисовки пропорциональна числу видимых точек, а не общей площади. Типичный символ 8x8 может иметь 20 установленных пикселей из 64. Для большого прокручиваемого сообщения эта экономия крайне важна. ВС указывает на данные шрифта; RLA сдвигает каждый бит во флаг переноса для определения — рисовать или нет.

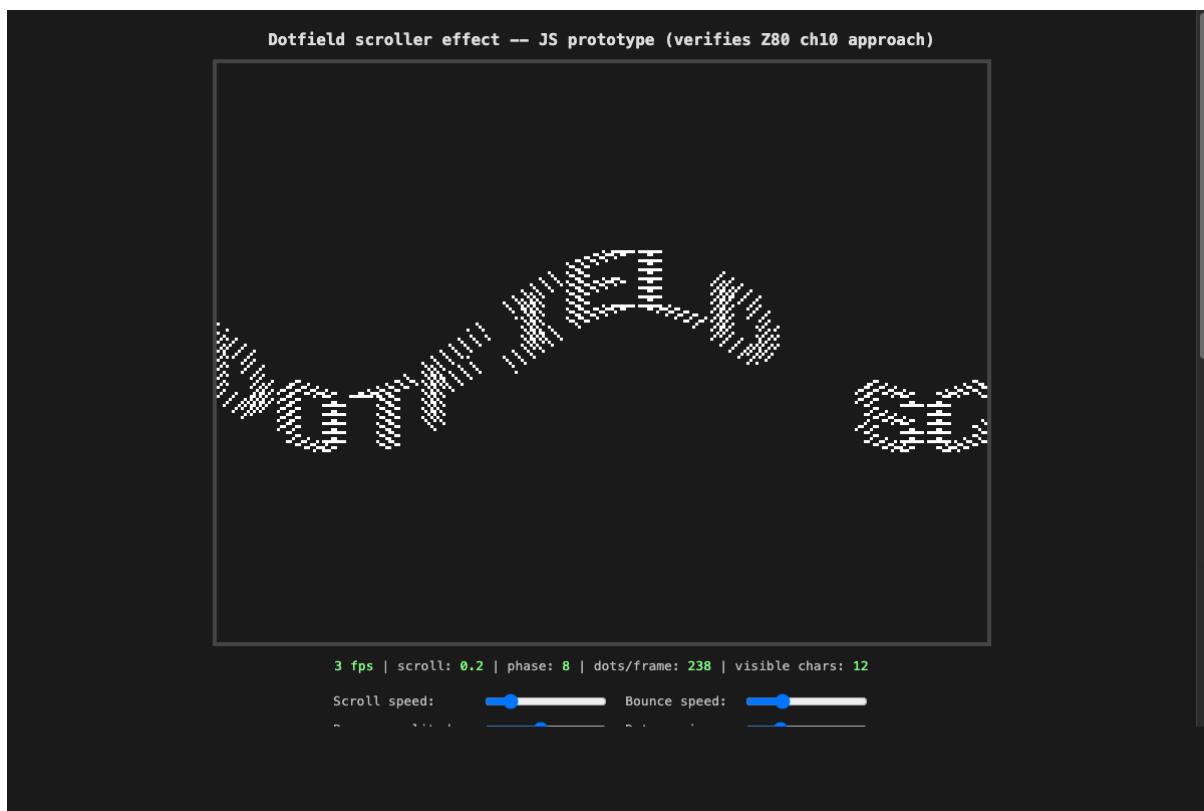


Рис. 23: Прототип скроллера с точечным полем — каждый символ отрисован как отдельные пиксели, прыгающие на синусоиде, что создаёт классический демосценовый текстовый эффект

Адресные таблицы на основе стека

В обычном скроллере экранная позиция каждого пикселя вычисляется из координат (x, y) по формуле чересстрочной адресации Spectrum. Это вычисление включает сдвиги, маски и обращения к таблицам. Делать это для тысяч пикселей за кадр поглотило бы весь бюджет кадра.

Решение Dark'a: предварительно вычислить каждый экранный адрес и сохранить их как таблицу, по которой проходит указатель стека. POP читает 2 байта и автоматически инкрементирует SP, всё за 10 тактов (T-states). Направь SP на

таблицу вместо реального стека, и POP становится самым быстрым способом извлечения адресов — ни индексных регистров, ни адресной арифметики, ни накладных расходов.

Сравни POP с альтернативами. LD A, (HL) : INC HL считывает один байт за 11 тактов — для считывания адреса понадобятся две такие пары (22 Т), плюс LD L,A / LD H,A для сборки. Индексированная загрузка вроде LD L, (IX+0) : LD H, (IX+1) стоит 38 тактов за пару. POP считывает оба байта, инкрементирует указатель и загружает регистровую пару — 10 тактов, без вариантов. Цена в том, что ты отдаёшь указатель стека рендереру. Ничто другое не может использовать SP, пока работает внутренний цикл.

Это значит, что прерывания фатальны. Если прерывание сработает, когда SP указывает в таблицу адресов, Z80 помещает адрес возврата в «стек» — а это на самом деле твоя таблица данных. Два байта тщательно вычисленных экранных адресов будут перезаписаны адресом возврата, и обработчик прерываний начнёт выполнять мусор, оказавшийся по испорченному адресу. Результат — отискажённого кадра до полного зависания. Решение простое и безоговорочное: DI перед захватом SP, EI после его восстановления. Каждая подпрограмма с POP-трюком в каждом спектрумовском демо следует этому паттерну:

```
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
```

Сохранение/восстановление использует самомодифицирующийся код (SMC), потому что это самый быстрый способ одновременно сохранить и восстановить SP. EX (SP), HL требует валидного стека. LD (addr), SP существует (опкод ED 73, 20 тактов), но она сохраняет SP по фиксированному адресу — потом понадобится отдельная LD SP, (addr) для восстановления (тоже 20 тактов), и восстановление не быстрее SMC-подхода. Техника SMC записывает значение SP прямо в поле операнда последующей инструкции LD SP, nnnn: LD (.smc+1), SP стоит 20 тактов на сохранение, а восстановление (LD SP, nnnn с подменённым операндом) — всего 10 тактов. Суммарные затраты на сохранение+восстановление составляют 30 тактов против 40 тактов для пары LD (addr), SP / LD SP, (addr) — небольшая экономия, которая к тому же не требует отдельной ячейки памяти.

Одно тонкое последствие: окно DI/EI блокирует кадровое прерывание. Если внутренний цикл работает долго, HALT в начале главного цикла всё равно поймает следующее прерывание — но если рендеринг выходит за пределы целого кадра, ты теряешь синхронизацию. Вот почему арифметика бюджета кадра так важна. Ты должен знать время наихудшего случая, прежде чем применять POP-трюк.

Подпрыгивающее движение целиком закодировано в адресной таблице. Каждая запись — экранный адрес, уже включающий вертикальное синусоидальное смещение. «Прыжок» не происходит во время отрисовки. Он произошёл, когда таблица была построена. Все три измерения анимации — позиция прокрутки, волна отскока, форма символа — сворачиваются в единую линейную последовательность 16-битных адресов, потребляемую на полной скорости через

POP.

Внутренний цикл

Анализ Introspec'a 2017 года раскрывает внутренний цикл Illusion. Один байт шрифтовых данных содержит 8 бит — 8 пикселей. LD A,(BC) читает байт один раз, затем RLA сдвигает по одному биту через 8 развёрнутых итераций:

```
; 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 ...
```

Стоимость на пиксель, без учёта амортизированного чтения байта:

Путь	Инструкции	Такты (T-states)
Непрозрач- ный пиксель	pop hl + rla + jr nc (не взят) + set ?,(hl)	36
Прозрачный пиксель	pop hl + rla + jr nc (взят)	26

LD A,(BC) и INC BC стоят 13 тактов (T-states), амортизированных на 8 пикселей — около 1,6 такта на пиксель. «36 тактов на пиксель» из анализа Introspec'a — это стоимость наихудшего случая в пределах развёрнутого байта, без учёта этих накладных расходов.

Позиция бита в SET меняется для каждого пикселя (7, 6, 5 ... 0), поэтому цикл развёрнут 8 раз, а не повторяется. Нельзя параметризовать позицию бита в SET без индексации через IX/IY (слишком медленно) или самомодифицирующегося кода (SMC) (лишние накладные расходы). Развёрнутый цикл — чистое решение.

Арифметика бюджета кадра

Давай посчитаем как следует. Стандартный кадр Spectrum 48K — 69 888 тактов (клон Pentagon работает чуть дольше — 71 680). Из них ULA крадёт такты во время активной области отображения из-за конкуренции за память, а скроллер



Рис. 24: Прыгающий точечный скроллер в действии — текст отрисован как отдельные пиксели, волнообразно движущиеся по синусоиде

пишет в экранную память в течение всего кадра, не только во время бордюра, поэтому конкуренция — реальный фактор. На практике считай, что доступно около 60 000 тактов на 48K и 65 000 на Pentagon. Вычти воспроизведение музыки (типичный AY-проигрыватель стоит 3000–5000 Т за кадр), очистку экрана и построение таблицы. Остаётся примерно 40 000–50 000 тактов на собственно отрисовку точек.

Рассмотрим отображение 8 символов шрифта 8x8 = 512 битов шрифта за кадр (8 символов × 8 байт × 8 бит). При типичном заполнении шрифта около 30% примерно 154 бита установлены (непрозрачные) и 358 — сброшены (прозрачные). Стоимость внутреннего цикла:

- 154 непрозрачных пикселя по 36 Т = 5 544 Т
- 358 прозрачных пикселей по 26 Т = 9 308 Т
- 64 считывания байтов (`LD A,(BC) : INC BC`) по 13 Т = 832 Т
- Итого: примерно 15 684 такта

Это с запасом укладывается в один кадр. Ты мог бы отрисовать 20+ символов, прежде чем упрёшься в потолок бюджета. Узкое место — не внутренний цикл, а построение таблицы. Создание 512 записей адресов с синусными подстановками и вычислением экранного адреса стоит примерно 100–150 тактов на запись (в зависимости от реализации), добавляя 50 000–75 000 Т к кадру. Illusion решает это предвычислением всего набора таблиц в памяти с циклическим перебором смещений, либо инкрементальным построением: когда скролл сдвигается на один пиксель, большинство записей таблицы сдвигаются на одну позицию, и только новый столбец требует полного пересчёта.

Цифры работают, потому что две оптимизации дают кумулятивный эффект. Адресация через стек устраняет все вычисления координат из внутреннего цикла. Текстурная прозрачность устраивает все записи для пустых пикселей. Построение таблицы затратно, но оно выполняется вне критичного по времени окна DI и может быть распределено по кадру.

Как закодирован отскок

Адресная таблица — это то место, где живёт искусство. Чтобы создать подпрыгивающее движение, таблица синусов смещает вертикальную позицию каждого столбца:

Два параметра частоты управляют визуальным характером волны. `phase_freq` определяет пространственную частоту — сколько волновых циклов укладывается в видимые столбцы точек. Значение 4 означает, что каждый столбец точек продвигается на 4 позиции в таблице синусов, так что $256/4 = 64$ столбца охватывают один полный цикл волны. Значение 8 удваивает частоту, создавая более плотную рябь. `speed_freq` управляет скоростью распространения волны во времени: большие значения заставляют колебание прокручиваться быстрее, независимо от прокрутки текста.

Сама таблица синусов — это 256-байтный массив знаковых смещений, выровненный по странице для быстрого доступа. Выравнивание по странице означает, что старший байт адреса таблицы фиксирован; меняется только младший, поэтому подстановка сводится к:

```
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
```

Значения в таблице знаковые: положительные смещения сдвигают точку вниз, отрицательные — вверх. Амплитуда заложена в таблицу при генерации. Таблица с диапазоном от -24 до +24 даёт колебание в 48 строк развертки от пика до пика. Генерация таблицы — одноразовая затрата, обычно выполняемая офлайн или при инициализации с помощью таблицы подстановки или простой аппроксимации. На Z80 вычисление истинных значений синуса в реальном времени дорого, поэтому демосценовые кодеры либо предвычисляют таблицы внешним инструментом, либо используют квадрантную симметрию: вычисляют четверть волны (64 записи), затем зеркально отражают и инвертируют знак, чтобы заполнить оставшиеся три четверти.

По координатам каждой точки ($x, y + y_{\text{offset}}$) вычисляется экранный адрес Spectrum и сохраняется в таблице. Код построения таблицы выполняется один раз за кадр, вне внутреннего цикла. Внутренний цикл видит лишь поток предварительно вычисленных адресов.

За пределами простого синуса: Лиссажу, спираль и многоволновые паттерны

Красота подхода с предвычисленной таблицей в том, что внутреннему циклу безразлично, какую форму описывает движение. Он потребляет адреса с фиксированной стоимостью независимо от траектории, которая их породила. Это делает тривиальным эксперименты с различными паттернами движения — вся сложность сосредоточена в коде построения таблицы.

Фигура Лиссажу добавляет горизонтальное синусное смещение в дополнение к вертикальному. Вместо того чтобы каждый столбец отображался на фиксированную x -позицию байта на экране, x -координата тоже осциллирует:

```
y_offset = sin_table[(column * y_freq + phase_y) & 255]
```

Когда `x_freq` и `y_freq` взаимно просты (скажем, 3 и 2), точечное поле вычерчивает фигуру Лиссажу — классический осциллографический паттерн. Текст превращается в ленту, вьющуюся в пространстве. Различные соотношения частот дают кардинально разные формы: 1:1 даёт круг или эллипс, 1:2 — восьмёрку, 2:3 — трилистник, знакомый по старому аналоговому измерительному оборудованию.

Эффект **спирали** или **геликоида** использует единственную фазу, нарастающую от столбца к столбцу, но с переменной амплитудой:

```
y_offset = sin_table[(column * freq + phase) & 255] * amplitude / max_amp
```

Это создаёт иллюзию точек, уходящих вглубь — волна сглаживается в «далней» точке спирали и расширяется в «ближней».

Многоволновая суперпозиция — простейшая техника с самым зреющим результатом. Сложи два синусных слагаемых с разными частотами:

Результат — сложная, органично выглядящая волна, которая практически не повторяется. Продвижение `phase1` и `phase2` с разными скоростями порождает непрерывно эволюционирующее движение всего из двух обращений к таблице на столбец. Три и более гармоники создают волны, которые выглядят почти как гидродинамика. Это самый дешёвый способ генерировать сложное движение — каждая дополнительная гармоника стоит одно обращение к таблице и одно сложение на столбец в построителе таблицы, а стоимость внутреннего цикла остаётся неизменной.

Часть 2: 4-фазная цветовая анимация

Проблема цвета

Каждая ячейка 8x8 имеет один цвет чернил (0-7) и один цвет бумаги (0-7). В пределах одного кадра ты получаешь ровно два цвета на ячейку. Но Spectrum работает на частоте 50 кадров в секунду, и человеческий глаз не различает отдельные кадры на такой частоте. Он видит среднее.

Трюк

4-фазная техника Introspec'a циклически переключается между четырьмя кадрами:

1. **Нормальный A:** `ink = C1, paper = C2`. Пиксельные данные = паттерн A.
2. **Нормальный B:** `ink = C3, paper = C4`. Пиксельные данные = паттерн B.
3. **Инвертированный A:** `ink = C2, paper = C1`. Пиксельные данные = паттерн A (те же пиксели, цвета поменяны местами).
4. **Инвертированный B:** `ink = C4, paper = C3`. Пиксельные данные = паттерн B (те же пиксели, цвета поменяны местами).

На частоте 50 Гц каждый кадр отображается в течение 20 миллисекунд. Четырёхкадровый цикл завершается за 80 мс — 12,5 циклов в секунду, выше порога слияния мерцания на CRT-дисплеях.

Математика восприятия

Проследим один пиксель, который «включён» в паттерне A и «выключен» в паттерне B:

Кадр	Состояние пикселя	Отображаемый цвет
Нормальный A	вкл (ink)	C1
Нормальный B	выкл (paper)	C4
Инвертированный A	вкл (ink)	C2
Инвертированный B	выкл (paper)	C3

Глаз воспринимает среднее: $(C1 + C2 + C3 + C4) / 4$.

Теперь проверим: пиксель, «включённый» в обоих паттернах, видит C1, C3, C2, C4. Пиксель, «выключенный» в обоих, видит C2, C4, C1, C3. Во всех случаях получается одно и то же среднее. Пиксельный паттерн не влияет на воспринимаемый оттенок — на него влияет только выбор C1 до C4.

Тогда зачем два паттерна? Потому что *промежуточные* переходы имеют значение. Пиксель, чередующийся между ярко-красным и ярко-зелёным, заметно мерцает на 12,5 Гц. Пиксель, чередующийся между близкими оттенками, едва мерцает. Паттерны дизеринга — шахматные, полутоновые, упорядоченные матрицы — управляет *текстурой* мерцания. Introspec подобрал паттерны так, чтобы переходы между кадрами давали минимальное видимое колебание. Это антиконфликтный подбор пикселей: тщательная расстановка «включённых» и «выключенных» битов, чтобы ни один пиксель не переключался между кардинально различающимися цветами в последовательных кадрах.

Почему инверсия необходима

Без шага инверсии «включённые» пиксели всегда показывали бы ink, а «выключенные» — всегда paper. Ты получил бы ровно два видимых цвета на ячейку, мерцающих между двумя разными парами. Инверсия гарантирует, что и ink, и paper вносят вклад в оба состояния пикселей на протяжении цикла, смешивая все четыре цвета в воспринимаемый результат.

На Spectrum инверсия обходится дёшево. Байт атрибутов имеет формат FBPPPIII — Flash, Bright, 3 бита фона (paper), 3 бита чернил (ink). Обмен ink и paper означает ротацию младших 6 бит: paper перемещается на позицию ink, ink — на позицию paper, а Flash и Bright остаются на месте. В коде:

```
; 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

```

Альтернатива — предвычислить оба буфера атрибутов (нормальный и инвертированный) при инициализации и просто переключать указатели буферов во время выполнения. Это обменивает 3072 байта памяти на нулевые вычисления за кадр — стоящий обмен на 128К-машинах, где памяти хватает.

Практическая стоимость

Четыре предварительно построенных буфера атрибутов, циклически сменяемых каждый кадр. Покадровая стоимость — блочное копирование 768 байт в область атрибутов (\$5800-\$5AFF). Через LDIR это стоит 21 такт на байт: $768 \times 21 = 16\,128$ тактов. С использованием стекового трюка (POP из исходного буфера, переключение SP, PUSH в атрибутную память, пакетирование через регистровые пары и теневые регистры) реалистичная стоимость — около 11 000-13 000 тактов, в зависимости от размера пакета и накладных расходов цикла — скромное ускорение в 1,2-1,5 раза по сравнению с LDIR. Выигрыш меньше, чем можно было бы ожидать, потому что каждый пакет требует двух переключений SP (сохранить позицию источника, загрузить адрес назначения, затем переключить обратно), и эти накладные расходы в значительной мере нивелируют преимущество POP+PUSH в чистой скорости перед LDIR. Для заливки (запись одного и того же значения в каждый байт) PUSH-трюк намного эффективнее — загрузи регистровые пары один раз и затем выполняй PUSH многократно — но копирование из различающихся исходных данных не может избежать затрат на чтение.

Логика циклической смены тривиальна. Единственная переменная хранит фазу (0-3). Каждый кадр: инкремент и AND с 3 для закольцовки. Индексация в 4-элементную таблицу базовых адресов буферов:

```

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

```

156 ГЛАВА 10: ТОЧЕЧНЫЙ СКРОЛЛЕР И 4-ФАЗНАЯ ЦВЕТОВАЯ АНИМАЦИЯ

Память: $4 \times 768 = 3072$ байта на буферы. На 48К-машине это значительная часть; на 128К можно разместить буферы в подключаемых банках памяти. Пиксельные паттерны (A и B) записываются один раз при инициализации и больше не затрагиваются — каждый кадр меняется только атрибутная память.

Текстовый оверлей

В Eager поверх цветовой анимации прокручивается текст. Существует несколько подходов, каждый со своими компромиссами.

Простейший — **исключение ячеек**: зарезервировать определённые символьные ячейки под текст, исключить их из цветового цикла и записать фиксированные атрибуты «белый на чёрном» с реальными глифами шрифта. Реализуется легко — достаточно замаскировать эти ячейки при копировании LDIR — но создаёт жёсткую визуальную границу между анимированным фоном и статичной текстовой областью. Текст выглядит наклеенным.

Более изощрённый подход — **интеграция в паттерны**: формы глифов переопределяют конкретные биты в обоих пиксельных паттернах A и B. Там, где у шрифта бит установлен, оба паттерна получают этот бит установленным (или сброшенным, в зависимости от желаемого цвета текста). Это гарантирует, что текстовый пиксель показывает один и тот же цвет во всех четырёх фазах — он не мерцает, потому что никогда не переключается между разными цветовыми состояниями. Окружающие пиксели продолжают циклическую смену в обычном режиме. Результат — текст, который как будто парит над анимированным фоном, с цветовым подтеканием к краям каждой буквы. Стоимость в том, что приходится регенерировать (или патчить) пиксельные паттерны при каждом сдвиге текста, что добавляет несколько тысяч тактов за кадр в зависимости от количества ячеек с текстом.

Третий вариант для 128К- машин — **послойное композитирование**: хранить 4-фазный фон в одном наборе страниц памяти, а текстовый скроллер — в другом, и комбинировать их при копировании атрибутов. Это делает две системы независимыми — скроллеру не нужно знать о цветовой анимации, и наоборот — ценой чуть более сложного цикла копирования, маскирующего текстовые ячейки.

Родословная на демосцене

Точечный скроллер не появился из ниоткуда. Техника стоит в линии эффектов ZX Spectrum, которая тянется с середины 1980-х до наших дней.

Самые ранние спектрумовские скроллеры были простыми посимвольными: горизонтальный скроллинг на LDIR, сдвигающий целую строку символьных ячеек по одному байту за раз. Плавный пиксельный скроллинг был сложнее — у Spectrum нет аппаратного регистра скроллинга, поэтому каждый сдвиг на пиксель требует перезаписи растровых данных. К началу 1990-х демосценовые кодеры разработали несколько подходов: пиксельный скроллинг на RL/RR (сдвиг каждого байта экранной строки), скроллеры на таблицах подстановки (предсдвинутые копии каждого символа) и технику двойной буферизации

(рисуем в задний буфер, копируем на экран). Все они были ограничены фундаментальной стоимостью перемещения байтов в видеопамять и из неё.

Точечный подход полностью разрывает с этой традицией. Вместо скроллинга сплошного блока пикселей он разбивает текст на отдельные точки и размещает каждую независимо. В этом состояла идея Dark в середине 1990-х: если отказаться от идеи сплошного шрифта и принять пуантилистский рендеринг, можно использовать POP-трюк для размещения каждой точки с минимальными накладными расходами. Визуальный результат — текст, распадающийся в облако частиц, прыгающих на синусоиде — стал одним из фирменных эффектов русской демосцены.

Illusion от X-Trade (ENLiGHT'96) было демо, которое прославило эту технику в мире Spectrum. Точечный скроллер был его центральным эффектом, плавно работающим наряду с музыкой и другими визуальными элементами. Dark опубликовал алгоритмические принципы в *Spectrum Expert* выпуски #01 и #02 (1997–98), где описал общий подход к POP-рендерингу и анимации на таблице синусов. Двадцать лет спустя детальный реверс-инжиниринг бинарника *Illusion*, выполненный Introspec (опубликован в журнале *Hype*, 2017), подтвердил заявления Dark и предоставил точные подсчёты тактов, о которых сообщество долго гадало.

Техника 4-фазного цвета имеет другую родословную. Цветовая циклическая смена на Spectrum исследовалась с 1980-х — простое двухкадровое чередование (эффекты типа *flash*) было обычным в играх и демо. Но систематический четырёхфазный подход, с его тщательным шагом инверсии, обеспечивающим равный вклад всех четырёх цветов, был доведён до совершенства Introspec для *Eager* (3BM Open Air 2015). В file_id.diz пати-версии техника упомянута явно, а статья Introspec «Making of Eager» в *Hype* (2015) описывает процесс проектирования: выбор цветов так, чтобы соседние фазы минимизировали видимое мерцание, и использование паттернов дизайна, равномерно распределенных переходы по ячейке.

Более широкий принцип — временное мультиплексирование цвета — встречается и на других платформах. Atari 2600 знаменита чередованием кадров для создания мерцающих псевдо-спрайтов. Game Boy использует похожий трюк для псевдо-прозрачности. На Spectrum техника особенно эффективна, потому что послесвечение люминофора ЭЛТ сглаживает переходы лучше, чем это сделал бы ЖК-дисплей. Стоит отметить для современных зрителей: 4-фазный цвет выглядит существенно лучше на настоящем ЭЛТ или в хорошем ЭЛТ-эмulyаторе (с симуляцией люминофора), чем на чистом попикельном дисплее.

Общий принцип: временной обман

Точечный скроллер использует 50 кадров в секунду для *пространственной* гибкости. Каждый кадр — это снимок позиций точек в один момент; мозг зрителя интерполирует между снимками, воспринимая плавное движение. Задача процессора — *расставить* точки как можно быстрее, читая предварительно вычисленные адреса из стека.

4-фазная цветовая анимация использует 50 кадров в секунду для цветовой гибкости. Каждый кадр показывает одно из четырёх цветовых состояний; сетчатка зрителя усредняет их. Ни один отдельный кадр не содержит воспринимаемого результата — он существует только в инерции зрительного восприятия.

Обе техники эксплуатируют одну и ту же физическую реальность: ЭЛТ обновляется на частоте 50 Гц, и зрительная система человека не в состоянии различить отдельные кадры на этой частоте. Временное разрешение Spectrum значительно богаче его пространственного или цветового разрешения. Демо-сценовые кодеры обнаружили, что временное разрешение — самая дешёвая ось для эксплуатации.

Обе техники сводят свои внутренние циклы к абсолютному минимуму. Скроллер — до 36 тактов на точку. Цветовая анимация — до одного копирования буфера за кадр. Обе выносят сложность из внутреннего цикла в предварительные вычисления. И обе дают результаты, которые, на взгляд обычного зрителя, кажутся невозможными для этого оборудования.

Вот что делает демосцену искусством времени. Скриншот точечного скроллера — россыпь пикселей. Скриншот 4-фазной цветовой анимации — два цвета на ячейку, ровно как предписывает аппаратура. Надо видеть их *в движении*, чтобы увидеть, как они работают. Красота — в последовательности, а не в отдельном кадре.

Практика 1: Текстовый скроллер с подпрыгивающей точечной матрицей

Построй упрощённый точечный скроллер: короткое текстовое сообщение, отрисованное как подпрыгивающее точечное поле с адресацией на основе РОР.

Структуры данных. Выровненный по странице растровый шрифт 8x8 (подойдёт ROM-шрифт по адресу \$3D00). 256-байтная таблица синусов для смещения отскока. RAM-буфер для адресной таблицы (до 4096 x 2 байт).

Построение таблицы. Перед каждым кадром итерируй по видимым символам. Для каждого бита в каждом байте шрифта вычисли экранный адрес с учётом синусоидального смещения и сохрани его в адресной таблице. Это выполняется один раз за кадр, вне внутреннего цикла.

Отрисовка. Запрети прерывания. Сохрани SP через самомодифицирующийся код (SMC). Направь SP на адресную таблицу. Выполни развёрнутый внутренний цикл: `ld a,(bc) : inc bc`, затем 8 повторений `pop hl : rla : jr nc,skip : set N,(hl)` с N от 7 до 0. Восстанови SP. Разреши прерывания.

Основной цикл. `halt` (синхронизация с 50 Гц), очистка экрана (PUSH-очистка из главы 3), построение адресной таблицы, отрисовка точечного поля, продвижение позиции прокрутки и фазы отскока.

Расширения. Частичная очистка экрана (отслеживание ограничивающего прямоугольника). Двойная буферизация через теневой экран на 128K. Множественные гармоники отскока. Переменная плотность точек для более разреженного, эфемерного вида.

Практика 2: 4-фазная анимация цветового цикла

Построй 4-фазную цветовую анимацию, создающую плавные градиенты.

Пиксельные паттерны. Заполни область раstra двумя комплементарными паттернами дизеринга. Простейший вариант: чётные пиксельные строки заполняются \$55 (01010101), нечётные — \$AA (10101010). Для продакшен-качества используй упорядоченную матрицу Байера 4x4.

Буферы атрибутов. Предварительно рассчитай четыре 768-байтных буфера. Буфера 0 и 1 содержат нормальные атрибуты с двумя различными цветовыми схемами (изменяющиеся ink/paper по экрану для диагонального градиента). Буфера 2 и 3 — инвертированные версии: биты ink и paper поменяны местами. Перестановка выполняется битовой ротацией: три RRCA для перемещения битов ink на позицию paper, три RLCA в обратном направлении, маска и объединение.

Основной цикл. Каждый кадр: halt, индекс в 4-элементную таблицу указателей на буфера с помощью счётчика фаз (AND 3), LDIR 768 байт в \$5800, инкремент счётчика фаз. Это весь движок времени выполнения — около 16 000 тактов (T-states) за кадр.

Анимация. Для движущегося градиента перегенерируй один буфер за кадр (тот, который скоро станет самым старым в 4-кадровом цикле) с продвигающимся цветовым смещением. Это поддерживает конвейер: отображай кадр N, генерируя кадр N+4. Альтернативно — предварительно рассчитай все буфера по банкам 128K для нулевой стоимости в рантайме.

Итого

- **Точечный скроллер** отрисовывает текст как отдельные точки. Внутренний цикл — pop hl : rla : jr nc,skip : set ?, (hl) — стоит 36 тактов (T-states) на непрозрачный пиксель, 26 — на прозрачный.
 - **Адресация через стек** кодирует траекторию отскока как предварительно построенные экранные адреса. POP извлекает их по 10 тактов за штуку — самое быстрое произвольное чтение на Z80.
 - **4-фазный цвет** циклически переключает 4 кадра атрибутов (2 нормальных + 2 инвертированных) на 50 Гц. Инерция зрительного восприятия усредняет цвета, создавая иллюзию более чем 2 цветов на ячейку.
 - **Шаг инверсии** гарантирует, что все четыре цвета вносят вклад в каждую пиксельную позицию.
 - Обе техники эксплуатируют **временное разрешение** для создания эффектов, невозможных в любом отдельном кадре.
 - Скроллер использует стек для пространственной гибкости; цветовая анимация использует чередование кадров для цветовой гибкости — две главные оси демосценового обмана.
-

Попробуй сам

1. Построй точечный скроллер. Начни с одного статичного символа, отрисованного через внутренний цикл на основе РОР. Проверь ожидаемые тайминги с помощью тестовой обвязки бордюра из главы 1. Затем добавь таблицу отскока и наблюдай, как он волнится.
2. Экспериментируй с параметрами отскока. Измени амплитуду синуса, пространственную частоту и скорость фазы. Небольшие изменения дают разительные визуальные различия.
3. Построй 4-фазную цветовую анимацию. Начни с однородного цвета (все ячейки одинаковые в каждой фазе). Убедись, что видишь ровный цвет, который не является ни ink, ни paper ни одного отдельного кадра. Затем добавь диагональный градиент.
4. Попробуй различные паттерны дизеринга. Шахматный, блоки 2x2, матрица Байера, случайный шум. Какие минимизируют видимое мерцание? Какие дают наиболее плавные воспринимаемые градиенты?
5. Объедини обе техники: 4-фазный цветной фон с монохромным точечным скроллером поверх.

Источники: Introspec, «Technical Analysis of Illusion by X-Trade» (Hype, 2017); Introspec, «Making of Eager» (Hype, 2015); Dark, «Programming Algorithms» (Spectrum Expert #01, 1997). Дизассемблирование внутреннего цикла и подсчёт тактов следуют анализу Introspec'a 2017 года. 4-фазная цветовая техника описана в making-of Eager и file_id.diz пати-версии.

Глава 11: Звуковая архитектура - AY, TurboSound и Triple AY

“У чипа AY три голоса. Это не ограничение – это конструктивное решение, сформировавшее целый музыкальный жанр.”

AY-3-8912 – голос ZX Spectrum 128K. General Instrument спроектировала семейство AY-3-8910 в 1978 году как программируемый генератор звука (PSG, Programmable Sound Generator) – единственный чип, способный воспроизводить музыку и звуковые эффекты, не загружая процессор генерацией звука. Investronica первой установила AY в испанский Spectrum 128K, Sinclair переняла это решение, а к моменту, когда Amstrad унаследовала его для +2/+3, чип уже был проверенной рабочей лошадкой: Intellivision, MSX, Atari ST и десятки аркадных автоматов использовали его или его pin-совместимые варианты (YM2149 в Atari ST). В Spectrum 128K используется именно AY-3-8912, у которого меньше портов ввода-вывода, чем у 8910, но звуковые возможности полностью идентичны.

Четырнадцать регистров. Три канала прямоугольной волны. Один генератор шума. Один генератор огибающей. Это всё, что ты получаешь. Всё, что ты когда-либо слышал в чиптюне для Spectrum 128K – качающие басовые линии, стремительные арпеджированные аккорды, резкие барабаны – создаётся программированием этих четырнадцати регистров пятьдесят раз в секунду.

Эта глава проведёт тебя от голых записей в регистры до работающего музыкального движка. К её концу ты будешь точно понимать, как проигрыватель трекера передаёт звук в AY, и будешь обладать знаниями, чтобы написать свой собственный.

11.1 Карта регистров AY-3-8910

AY имеет 14 регистров, адресуемых R0-R13. На ZX Spectrum 128K доступ к ним осуществляется через два порта ввода-вывода:

- **\$FFFD** – выбор регистра (сначала записываешь сюда номер регистра)
- **\$BFFD** – запись данных (затем записываешь сюда значение)

Всегда сначала выбирай регистр, затем записывай данные. Этот двухшаговый процесс фундаментален:

```
; 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
```

Обрати внимание на трюк: мы меняем только старший байт BC между двумя инструкциями OUT. Младший байт \$FD остаётся в C на протяжении всего вызова. Это экономит повторную загрузку полного 16-битного значения.

Полная таблица регистров

Рег.	Название	Биты	Описание
R0	Период тона A, младший	8	Младшие 8 бит периода тона канала A
R1	Период тона A, старший	4	Старшие 4 бита периода тона канала A
R2	Период тона B, младший	8	Младшие 8 бит периода тона канала B
R3	Период тона B, старший	4	Старшие 4 бита периода тона канала B
R4	Период тона C, младший	8	Младшие 8 бит периода тона канала C
R5	Период тона C, старший	4	Старшие 4 бита периода тона канала C
R6	Период шума	5	Период генератора шума (0-31)
R7	Микшер / разрешение I/O	8	Разрешение тона и шума по каналам + I/O
R8	Громкость A	5	Громкость канала A (0-15) или режим огибающей
R9	Громкость B	5	Громкость канала B (0-15) или режим огибающей
R10	Громкость C	5	Громкость канала C (0-15) или режим огибающей
R11	Период огибающей, младший	8	Младшие 8 бит периода огибающей
R12	Период огибающей, старший	8	Старшие 8 бит периода огибающей
R13	Форма огибающей	4	Форма волны огибающей (0-15)

AY-3-8910 Register Map

R0	Channel A Tone Period Fine	7-0: Fine tune
R1	Channel A Tone Period Coarse	3-0: Coarse tune
R2	Channel B Tone Period Fine	7-0: Fine tune
R3	Channel B Tone Period Coarse	3-0: Coarse tune
R4	Channel C Tone Period Fine	7-0: Fine tune
R5	Channel C Tone Period Coarse	3-0: Coarse tune
R6	Noise Period	4-0: Period (0-31)
R7	Mixer Control	Expanded below
R7 Mixer Bit Layout:		
	bit 7 bit 6 bit 5 bit 4 bit 3 bit 2 bit 1 bit 0	
	IOB IOA NoiseC NoiseB NoiseA ToneC ToneB ToneA	
0 = ON (active low!) 1 = OFF / disabled		
R8	Channel A Volume	4: M (env), 3-0: Volume
R9	Channel B Volume	4: M (env), 3-0: Volume
R10	Channel C Volume	4: M (env), 3-0: Volume
R11	Envelope Period Fine	7-0: Fine
R12	Envelope Period Coarse	7-0: Coarse
R13	Envelope Shape	3-0: Shape (0-15)
R14	I/O Port A	7-0: Data
R15	I/O Port B	7-0: Data



Рис. 25: AY-3-8910 register map

Тональные каналы (R0-R5): как работает высота тона

Каждый из трёх тональных каналов генерирует прямоугольную волну. Частота контролируется 12-битным значением периода, разделённым на два регистра:

На Spectrum 128K тактовая частота AY составляет 1,7734 МГц (половина тактовой частоты ЦП 3,5469 МГц). Так, для ноты до первой октавы (примерно 262 Гц):

12-битный период даёт диапазон от 1 (110 837 Гц – неслышимый ультразвук) до 4095 (27 Гц – глубокий басовый гул). Вот практическая таблица нот для октавы 4:

Нота	Частота (Гц)	Период (дес.)	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
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

Чтобы подняться на октаву, раздели период пополам. Чтобы опуститься – удвой. Полная таблица нот для всех полезных октав находится в Приложении G.

Генератор шума (R6)

Регистр R6 управляет единственным генератором шума, общим для всех трёх каналов. 5-битное значение (0-31) задаёт “высоту” шума – меньшие значения дают более высокий шипящий шум; большие значения – более низкий грубый шум. Генератор шума производит псевдослучайный выход с использованием 17-битного регистра сдвига с линейной обратной связью.

Полезные диапазоны: - **0-5**: Высокий шип (хай-хэт, тарелка) - **6-12**: Средний шум (тело малого барабана) - **13-20**: Низкий рокот (взрыв) - **21-31**: Очень низкий (ветер, гром)

R7: Микшер – самый важный регистр

Регистр R7 – сердце AY. Шесть бит управляют тем, какие каналы получают тон, шум или и то, и другое. Ещё два бита управляют направлением портов ввода-вывода (не имеет значения для звука, оставляй их как входы = 1).

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)

Запутывающая часть: 0 означает ВКЛ. На этом спотыкаются все при первом знакомстве. AY использует логику с активным низким уровнем для разрешений микшера. Сброшенный бит включает соответствующий источник.

Вот таблица полезных комбинаций микшера:

Значение	Двоичное (NcNbNa TcTbTa)	Эффект
\$38	111 000	Все три тона включены, без шума
\$3E	111 110	Только тон А
\$3D	111 101	Только тон В
\$3B	111 011	Только тон С
\$36	110 110	Тон А + Шум А
\$07	000 111	Все три шума, без тонов
\$30	110 000	Тоны А+В+С, Шум А
\$00	000 000	Всё включено (акофония)
\$3F	111 111	Всё выключено (тишина)

Типичный паттерн для музыки: тоны А+В для мелодии и гармонии, тон С + шум С для ударных:

```
; 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
```

Регистры громкости (R8-R10)

Каждый канал имеет 4-битное управление громкостью (0-15, где 15 – самая громкая). Но бит 4 особенный: его установка переключает канал в **режим огибающей**, где громкость контролируется автоматически генератором огибающей вместо фиксированного значения.

Bit 4: 1 = use envelope generator, 0 = use bits 3-0
 Bits 3-0: Fixed volume level (0-15)

Кривая громкости логарифмическая на настоящем AY-3-8910 (каждый шаг составляет приблизительно 1,5 дБ), но варьируется между ревизиями чипа и клонами. YM2149 имеет другую кривую громкости, поэтому одна и та же мелодия звучит немного по-разному на Atari ST и на Spectrum.

Генератор огибающей (R11-R13)

AY имеет один генератор огибающей – общий для всех каналов, которые его используют. Он автоматически модулирует громкость согласно повторяющейся форме волны.

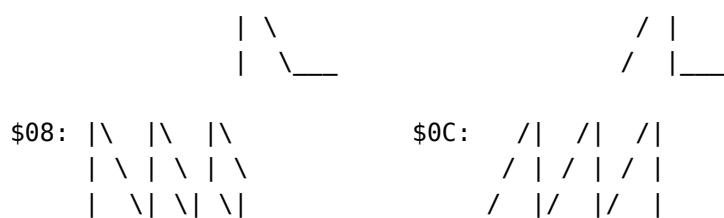
R11-R12: Период огибающей – 16-битное значение, управляющее скоростью огибающей:

При периоде = 1 огибающая циклится с частотой около 6 927 Гц. При периоде = 65535 она циклится примерно 0,11 Гц – приблизительно один раз каждые 9 секунд.

R13: Форма огибающей – четыре бита выбирают форму волны. Хотя 4 бита допускают 16 значений, уникальных форм только 10:

Значение	Форма	Описание
\$00-\$03	__	Однократное затухание, затем тишина. (Все четыре идентичны.)
\$04-\$07	/__	Однократная атака, затем тишина. (Все четыре идентичны.)
\$08	__\\	Повторяющаяся пила вниз.
\$09	____	Однократное затухание, затем тишина. (То же, что \$00.)
\$0A	\/__/_	Повторяющийся треугольник (затухание-атака).
\$0B	__\\'	Однократное затухание, затем удержание на максимуме.
\$0C	__\\/__	Повторяющаяся пила вверх.
\$0D	/'___	Однократная атака, затем удержание на максимуме.
\$0E	/_\\/__	Повторяющийся треугольник (атака-затухание).
\$0F	/__	Однократная атака, затем тишина. (То же, что \$04.)

Диаграммы форм волн (громкость во времени):



\$0A: \ \ / \ \ / \ \ / \ \	\$0E: / \ \ / \ \ / \ \ / \ \
\$0B: \ \ / \ \	\$0D: / \ \ / \ \

AY-3-8910 Envelope Shapes (R13)

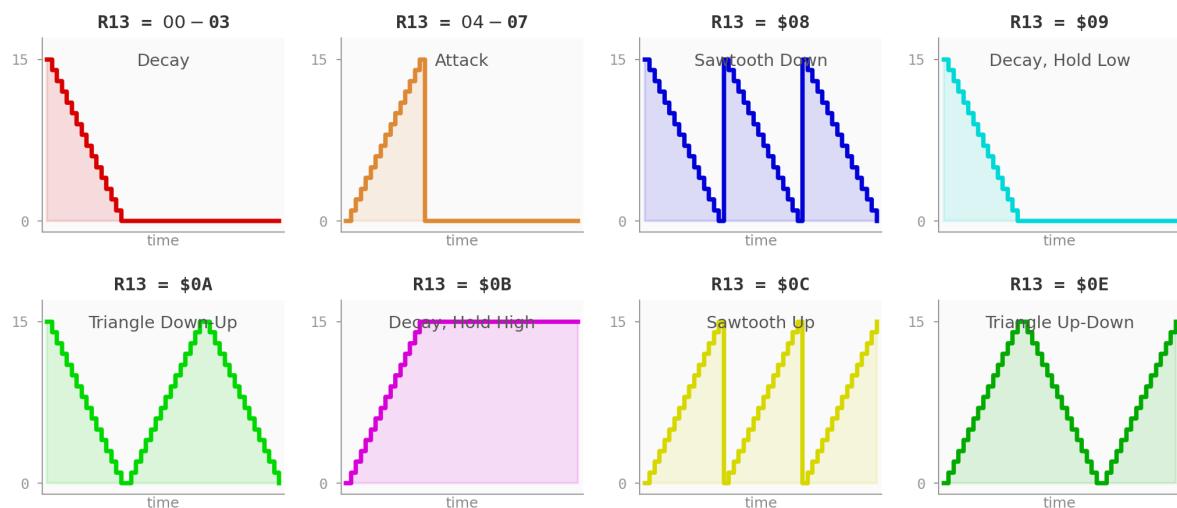


Рис. 26: AY envelope shape waveforms

Ключевое понимание: Запись в R13 *перезапускает* огибающую с начала. Это критически важно для басовых техник – ты можешь запустить новый цикл огибающей в любой момент, записав в R13, даже с тем же значением.

11.2 Чиптюн-техники на 3 каналах

Три канала – это немного. У настоящей группы как минимум бас, ударные, мелодия и гармония. Чиптюн-композиторы разработали изобретательные трюки для создания иллюзии большего:

Арпеджио: имитация аккордов

Арпеджио быстро перебирает ноты аккорда – по одной ноте на кадр. При 50 Гц (PAL) перебор C4, E4 и G4 каждый кадр создаёт эффект, который ухо воспринимает как аккорд до-мажор, хотя в каждый момент звучит только одна нота.

```
; 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)
inc hl
ld d, (hl)

; Write period to channel A
ld a, R_TONE_A_L0
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

```

В трекере арпеджио записывается как эффект, применяемый к нотам. Vortex Tracker II использует таблицы орнаментов, задающие покадровые смещения в полутонах.

Buzz-bass: трюк с огибающей

Вот самый характерный чиптюн-басовый звук: “базз”. Он работает через злоупотребление генератором огибающей. Устанавливаешь огибающую на короткую повторяющуюся пилю (\$08 или \$0C), устанавливаешь период огибающей, соответствующий желаемой частоте басовой ноты, и перезапускаешь огибающую каждый кадр. Результат – жужжащий, плотный басовый тон, совершенно не похожий на простую прямоугольную волну.

```

; Buzz-bass: play bass note using envelope generator
; DE = envelope period for desired note
buzz_bass:
    ; Set envelope period
    ld a, R_ENV_L0
    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

```

Период огибающей для заданной басовой ноты:

Для баса С2 (65,4 Гц): period = 1 773 400 / (256 x 65,4) = 106.

Buzz-bass даёт тебе басовый инструмент, звучащий принципиально иначе, чем тональные каналы, фактически добавляя четвёртый голос в аранжировку.

Проблема выравнивания периодов

Есть подвох с buzz-bass, который равномерно-темперированные таблицы нот скрывают от тебя. Посмотри на формулу ещё раз:

```
envelope_period = tone_period / 16
```

Чтобы базз звучал чисто, период огибающей должен быть *точно* tone_period / 16. Но целочисленное деление обрезает. Если период тона не делится на 16, период огибающей имеет ошибку округления – и форма волны огибающей дрейфует относительно тона, создавая слышимые биения.

Проверим нашу стандартную таблицу. Октава 4:

Нота	Период	Период mod 16	Огибающая = Период / 16	Ошибка?
C4	424	8	26 (должно быть 26.5)	Да
D4	378	10	23 (должно быть 23.625)	Да
E4	337	1	21 (должно быть 21.0625)	Да
F4	318	14	19 (должно быть 19.875)	Да
G4	283	11	17 (должно быть 17.6875)	Да
A4	252	12	15 (должно быть 15.75)	Да
B4	225	1	14 (должно быть 14.0625)	Да

Ни одного чистого деления! Каждая нота в равномерно-темперированной шкале даёт слегка расстроенную огибающую. Для коротких перкуссионных басс-звуков биения маскируются, но для протяжных басовых нот они создают неприятное дрожание.

Натуральный строй: Таблица #5

В июне 2001 года Ivan Roshin опубликовал “Частотная таблица с нулевой погрешностью”, прида к тому же выводу, который столетия музыкальной теории уже установили: заменить равномерную темперацию на *натуральный строй* – целочисленные интервалы, которые аппаратура AY может делить чисто.

Натуральная шкала для до-мажор / ля-минор использует следующие интервалы:

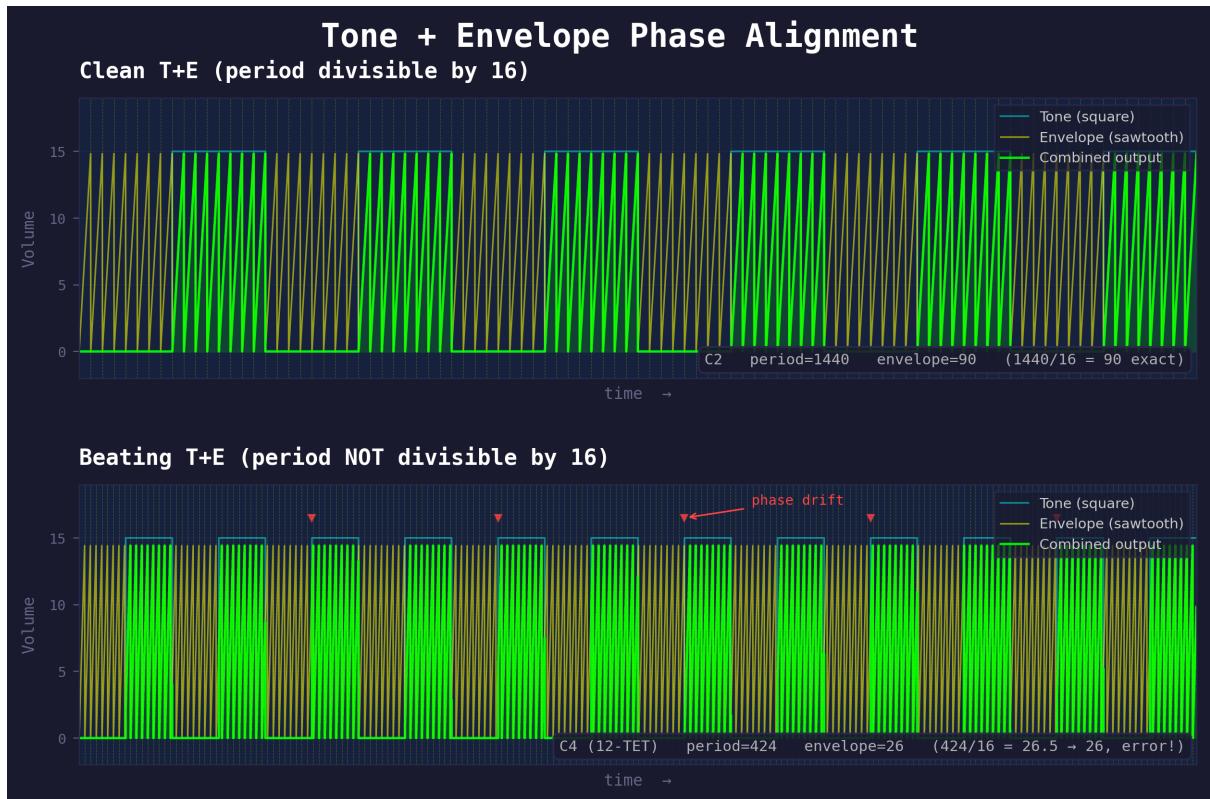


Рис. 27: Tone + Envelope Phase Alignment: clean T+E with period divisible by 16 (top) vs beating T+E with rounding error (bottom)

Это даёт чистые квинты (соотношение 3:2) для C-G, E-B, A-E. Хроматические ноты (диезы/бемоли) рассчитываются с соотношением 16/15.

Полученные периоды, рассчитанные для *нестандартной* тактовой частоты AY 1 520 640 Гц:

```
; 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
```

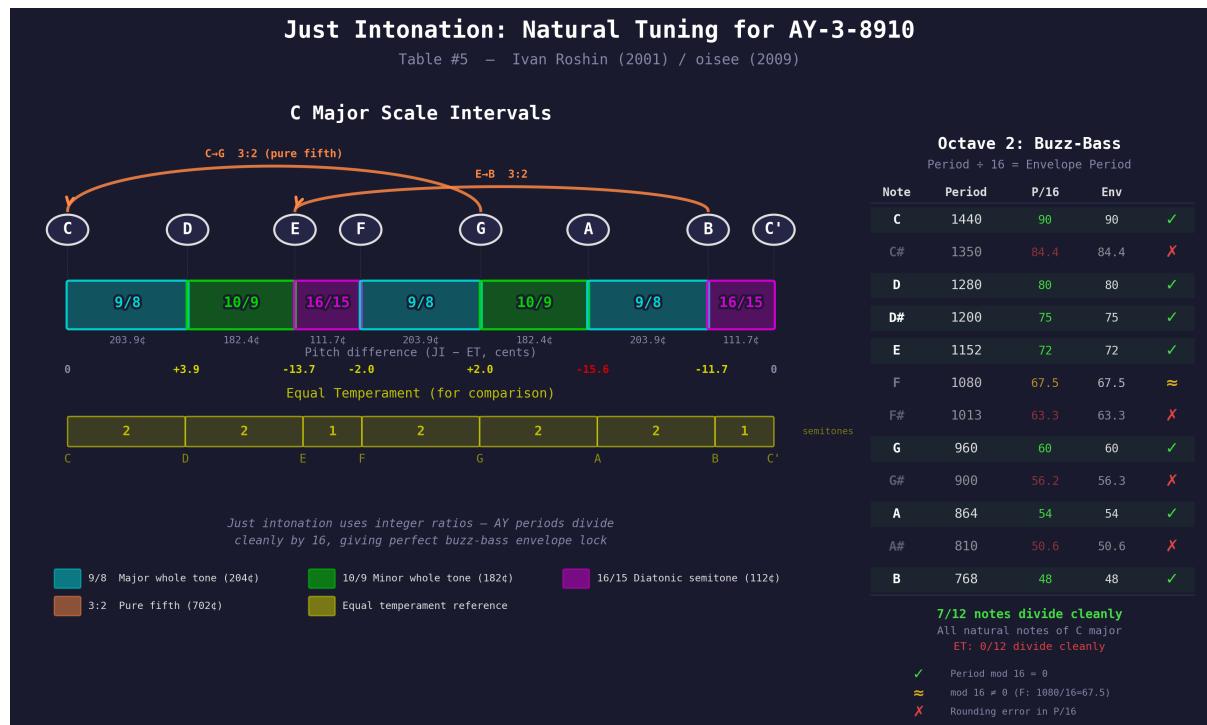


Рис. 28: Just Intonation: interval structure of the natural scale with period divisibility table for buzz-bass

```
; 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
```

Ключевое понимание в том, что большинство периодов основной шкалы теперь делятся на 16. Вот октава 2 – басовый диапазон, наиболее важный для базза:

Нота	Период	mod 16	Огибающая = Период/16	Чисто?
C2	1440	0	90	Да
C#2	1350	6	84.375 → 84	Нет
D2	1280	0	80	Да
D#2	1200	0	75	Да
E2	1152	0	72	Да
F2	1080	8	67.5 → 68	~
F#2	1013	5	63.3 → 63	Нет
G2	960	0	60	Да
G#2	900	4	56.25 → 56	Нет
A2	864	0	54	Да
A#2	810	10	50.6 → 51	Нет
B2	768	0	48	Да

Нота	Период	mod 16	Огибающая = Период/16	Чисто?
C2	1440	0	90	Да
C#2	1350	6	84.375 → 84	Нет
D2	1280	0	80	Да
D#2	1200	0	75	Да
E2	1152	0	72	Да
F2	1080	8	67.5 → 68	~
F#2	1013	5	63.3 → 63	Нет
G2	960	0	60	Да
G#2	900	4	56.25 → 56	Нет
A2	864	0	54	Да
A#2	810	10	50.6 → 51	Нет
B2	768	0	48	Да

Семь из двенадцати нот делятся чисто – все натуральные ноты до-мажора. Сравни с равномерно-темперированной таблицей, где *ни одна* не делится. На этих семи нотах генераторы огибающей и тона синхронизируются по фазе, и buzz-bass звучит чисто.

Компромисс: эта таблица верна только для до-мажор / ля-минор. Чтобы играть в других тональностях, нужно менять тактовую частоту AY:

Тональность	Частота чипа (Гц)
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

На реальном оборудовании тактовая частота AY фиксирована, поэтому ты не можешь менять тональности в рантайме. Но в эмуляторе или трекере вроде Vortex Tracker II “частота чипа” – это настройка. Именно это сделала модификация Vortex Tracker Improved (VTi) от oisee в 2009 году: она добавила Таблицу #5 (пятую таблицу, индекс 4, считая с нуля) с этими натуральными периодами, плюс настройку частоты чипа для каждого модуля, выбирающую тональность.

Конвертер autosiril MIDI-to-PT3 по умолчанию использует Таблицу #5 именно из-за этих чистых соотношений огибающей – большинство сконвертированных треков активно используют buzz-bass, и натуральный строй устраняет биения.

На практике: если ты пишешь трекерный модуль с интенсивным использованием buzz-bass, подумай о композиции в C/Am с Таблицей #5. Огибающие идеально синхронизируются с тоном. Если нужна другая тональность – либо

транспонируй частоту чипа (на стороне трекера), либо смирись с небольшими ошибками округления равномерной темперации. Для коротких перкуссионных базз-звуков разница неслышима; для протяжных басовых дронов она очень заметна.

Синтез ударных

С единственным генератором шума ударные должны делить его. Стандартный подход:

```
; 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_L0
    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

; 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_L0
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: 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_L0
    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

; 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_L0
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

```

Хай-хэт: очень короткий шумовой всплеск, высокая частота шума (низкое значение R6), немедленное снижение громкости после 1-2 кадров.

Орнаменты: покадровая модуляция

```

; 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

; 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

```

11.3 TurboSound: 2 x AY

Pentagon и Scorpion-клоны представили TurboSound – два чипа AY в одной машине. Второй чип адресуется через битовый паттерн, записываемый в порт \$FFFD перед операциями с регистрами.

Выбор чипа

```

; Select chip 0 (primary)
ld   bc, $FFFD

```

```

ld  a, $FF          ; bit pattern: select chip 0
out (c), a

; Select chip 1 (secondary)
ld  bc, $FFFFD
ld  a, $FE          ; bit pattern: select chip 1
out (c), a

; Select chip 0 (primary)
ld  bc, $FFFFD
ld  a, $FF          ; bit pattern: select chip 0
out (c), a

; Select chip 1 (secondary)
ld  bc, $FFFFD
ld  a, $FE          ; bit pattern: select chip 1
out (c), a

```

После выбора чипа все последующие чтения и записи регистров через *FFFFD/BFFD* идут к этому чипу. На практике твой музыкальный движок выбирает чип 0, обновляет все его регистры, затем выбирает чип 1 и обновляет его регистры.

6 каналов, настоящее стерео

TurboSound удваивает всё: 6 тональных каналов, 2 независимых генератора шума, 2 независимых генератора огибающей. Типичная стерео-расстановка:

Чип	Каналы	Стерео	Роль
Чип 0	A0, B0, C0	Лево или центр	Ведущая мелодия, гармония, бас
Чип 1	A1, B1, C1	Право или центр	Контрмелодия, пэды, ударные

Или смешай их для широкого стерео-поля: - Бас на обоих чипах (центр) - Мелодия на чипе 0 (лево) - Контрмелодия на чипе 1 (право) - Ударные распределены: бочка на чипе 0, малый/хай-хэт на чипе 1

Что TurboSound меняет в музыкальном плане – значительно: на одном AY композитор постоянно идёт на жертвы. Невозможно одновременно иметь протяжную басовую ноту, ведущую мелодию и удар барабана без кражи каналов. С TurboSound у тебя есть пространство. Выделенный басовый канал, выделенные ударные и четыре оставшихся голоса для мелодии и гармонии. Эра компромиссов заканчивается.

Модификация движка

```

music_frame:
    ; Update chip 0
    ld  a, $FF
    ld  bc, $FFFFD
    out (c), a          ; select chip 0
    call update_chip0   ; write 14 registers

```

```
; Update chip 1
ld a, $FE
ld bc, $FFFFD
out (c), a           ; select chip 1
call update_chip1    ; write 14 registers
ret
```

Цикл записи регистров записывает все 14 регистров из буфера в ОЗУ. Для двух чипов ты поддерживаешь два 14-байтных буфера и выгружаешь их последовательно. Каждая запись регистра требует выбора регистра (LD A,reg + OUT), затем записи значения (LD A,val + OUT), что стоит около 36 тактов (T-state) на регистр. На 28 регистров суммарно: примерно 1 008 тактов, плюс накладные расходы на выбор чипа.

Цикл записи регистров записывает все 14 регистров из буфера в ОЗУ. Для двух чипов ты поддерживаешь два 14-байтных буфера и выгружаешь их последовательно. Каждая запись регистра требует выбора регистра (LD A,reg + OUT), затем записи значения (LD A,val + OUT), что стоит около 36 тактов (T-state) на регистр. На 28 регистров суммарно: примерно 1 008 тактов, плюс накладные расходы на выбор чипа.

11.4 Triple AY на ZX Spectrum Next

ZX Spectrum Next идёт дальше: три AY-совместимых звуковых чипа, дающие 9 каналов. Но реализация Next выходит за рамки простого устроения.

Расширенные возможности

Чипы AY на Next включают **панорамирование каждого канала**. Каждый канал может быть индивидуально распределён влево, вправо или по центру – то, чего оригинальный AY никогда не поддерживал. Это управляется через дополнительные регистры, специфичные для Next.

Три чипа адресуются через регистр Next \$06 (peripheral 2 setting) или через стандартный порт \$FFFF со значениями выбора чипа.

9 каналов: оркестровое мышление

Девять каналов принципиально меняют подход к композиции на 8-битном оборудовании. Вместо хитрых трюков для имитации сложности ты можешь мыслить оркестрово:

Этого достаточно для по-настоящему богатых аранжировок. У тебя есть выделенный канал SFX, который никогда не прерывает музыку. У тебя есть независимые генераторы шума для многослойной перкуссии. Ты можешь держать аккорды без арпеджио. Характер AY сохраняется – прямоугольные волны остаются прямоугольными – но композиционная свобода приближается к свободе Amiga MOD-трекера с его четырьмя сэмплерными каналами.

11.5 Архитектура музыкального движка

Музыкальный движок – это код, который читает данные паттернов и записывает регистры AY в правильном темпе. На Spectrum он живёт внутри обработчика прерываний.

```
; 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
```

IM2 (Interrupt Mode 2) ZX Spectrum срабатывает раз в кадр – каждую 1/50 секунды на PAL-системах. Музыкальный движок подключается к нему:

```
; 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

```

Цикл проигрывателя

Подпрограмма проигрывателя, вызываемая 50 раз в секунду, выполняет следующее:

```

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

```

Упрощённый каркас:

```

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

```

Бюджет кадра

Вот критический вопрос: сколько тактов (T-state) может потреблять музыкальный движок, прежде чем он оставит основную программу голодной?

Кадр составляет 71 680 тактов (T-state) на Pentagon (69 888 на 48K). Прерывание срабатывает в начале кадра. Если музыкальный проигрыватель занимает 5 000 тактов, основной программе остаётся 66 680 на визуальные эффекты.

Типичные затраты: - **Простой проигрыватель** (без эффектов, без орнаментов): ~1 500–2 500 тактов (T-state) - **Проигрыватель Pro Tracker 3**: ~3 000–5 000 тактов (T-state) - **Полный проигрыватель Vortex Tracker II** с орнаментами и эффектами: ~4 000–7 000 тактов (T-state) - **Проигрыватель TurboSound** (2 чипа): ~6 000–10 000 тактов (T-state)

Для демо с ресурсоёмким эффектом 7 000 тактов на музыку – это существенно – около 10% кадра. Планируй соответственно.

Форматы и трекеры

Современный стандарт для AY-музыки на Spectrum – **Vortex Tracker II** (формат .pt3). Это кроссплатформенный трекер, работающий под Windows и выдающий файлы, непосредственно воспроизводимые проверенными Z80-проигрывателями.

Формат	Трекер	Возможности	Размер проигрывателя
.pt3	Vortex Tracker II / Pro Tracker 3	Орнаменты, сэмплы, эффекты	~1.2-1.8 КБ
.asc	ASC Sound Master	Проще, меньший проигрыватель	~0.8-1.0 КБ
.sqt	SQ-Tracker	Компактный, хорошее сжатие	~0.6-0.8 КБ
.stc	Sound Tracker	Базовый, самый старый	~0.5-0.7 КБ

```
; 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"
```

1. Сочиняешь в Vortex Tracker II на PC
2. Экспортируешь как .pt3 файл
3. Включаешь исходник .pt3 проигрывателя (Z80 ассемблер) в свой проект
4. Включаешь файл данных .pt3 как бинарный блоб
5. Вызываешь music_init при запуске (передавая адрес данных песни)
6. Вызываешь music_play из обработчика прерываний каждый кадр

Это стандартный подход, используемый практически каждым демо и игрой для Spectrum 128K с начала 2000-х.

11.6 Система звуковых эффектов

```
; 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_L0
    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

```

Кражка каналов

```

; 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_L0
    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

```

Процедурные таблицы SFX

```

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

```

Взрыв: шум с затухающей громкостью.

```

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

```

Лазер: быстрый свип тона вниз.

```

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

```

Прыжок: короткий свип тона вверх.

```
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
```

```
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 Собираем всё вместе: рабочий пример

Файл `chapters/ch11-sound/examples/ay_test.a80` содержит полный, компилирующийся пример, демонстрирующий основы: инициализацию AY, настройку микшера, запись периодов тона и воспроизведение мелодии. Изучай его параллельно с этой главой – каждая рассмотренная здесь концепция применяется в этом коде.

Ключевые паттерны, на которые стоит обратить внимание в примере:

Чтобы расширить этот пример до настоящего музыкального проигрывателя, нужно заменить линейную таблицу мелодии на данные на основе паттернов, добавить обработку орнаментов и перенести воспроизведение в обработчик прерываний IM2, чтобы основной цикл был свободен для визуальных эффектов.

Врезка: Beep - краткая история невозможного

До 128K и его чипа AY оригиналный 48K Spectrum имел ровно один бит аудиовыхода. Пин 4 порта \$FE. Высокий или низкий. Вот и всё.

Один бит означает одну прямоугольную волну с частотой, с которой ты его переключаешь. Никакого контроля громкости, никакого микширования, никакой аппаратной помощи. Чтобы воспроизвести ноту, ты сидишь в тесном цикле, переключая бит на нужной частоте. Чтобы

воспроизвести две ноты, ты чередуешь два цикла переключения. Чтобы три – чередуешь три. Каждый дополнительный голос поглощает процессорное время, которое могло бы тратиться на что-то другое – скажем, на рисование графики.

И тем не менее.

Между 2010 и 2015 годами Shiru (Shiru Otaku) каталогизировал примерно 30 различных движков бипера, каждый из которых использовал свою технику для извлечения многоголосия из одного бита. Подходы варьировались от простого чередования пульсов с pin-совместимостью (2-3 канала) до выдающихся инженерных подвигов:

- **ZX-16** от Jan Deak: 16-канальное многоголосие на одном бите. Шестнадцать. Процессор не делает ничего, кроме переключения бита динамика по тщательно рассчитанному расписанию, аппроксимирующему сумму 16 независимых волновых форм через импульсно-плотностную модуляцию.
- **Octode XL**: 8-канальный движок бипера, который действительно оставляет достаточно ЦП для визуальных эффектов.
- **Rain** от Life on Mars (2016): полноценное демо, работающее с 9-канальным движком бипера *одновременно с визуальными эффектами* на 48K Spectrum. Вся постановка – музыка и графика – работает без чипа AY, без переключения банков, без чего-либо сверх базовой машины.

Это одни из самых выдающихся инженерных достижений в 8-битных вычислениях. Они доказывают, что ограничения проницаемее, чем кажется. Но они также непрактичны для общего применения: большинство движков бипера потребляют 50-90% процессорного времени, почти ничего не оставляя для геймплея или эффектов. Чип AY существует именно для того, чтобы разгрузить генерацию звука на выделенное оборудование.

Мы рассматриваем здесь движки бипера как исторический контекст и вдохновение. Для практической музыки в демо и играх AY – то, на чём стоит сосредоточить усилия.

Врезка: Agon Light 2 - звуковая система VDP

Agon Light 2 использует совершенно другой подход к звуку. Его аудио генерируется сопроцессором ESP32 (VDP), а не выделенным звуковым чипом. Ты отправляешь VDU-команды по последовательной связи, и ESP32 синтезирует аудио программно.

Формы волн: звуковая система Agon предлагает несколько типов форм волн на каждый канал – прямоугольная, синусоидальная, треугольная, пилообразная и шум. Это уже гибче, чем генераторы тона AY, ограниченные прямоугольной волной.

ADSR-огибающие: каждый канал имеет полностью программируемую огибающую Attack-Decay-Sustain-Release. Без совместного ис-

пользования – каждый канал получает свою независимую огибающую, в отличие от единственного общего генератора огибающей AY.

Количество каналов: звуковая система VDP поддерживает несколько одновременных каналов (точное число зависит от версии прошивки, но обычно 8 и более).

Компромисс: звук Agon управляется через VDU-байтовые последовательности, отправляемые по последовательной связи. Это означает:

- Более высокую задержку, чем прямая запись в регистры (время последовательной передачи) - Менее точный тайминг (невозможно bit-bang-ить синхронизацию с точностью до такта) - Но значительно меньшую нагрузку на ЦП (eZ80 просто отправляет команды; ESP32 делает весь синтез)

Парадигма ближе к MIDI, чем к программированию на уровне регистров. Ты говоришь VDP “воспроизведи эту ноту на канале 3 синусоидальной волной с такой ADSR-огибающей”, и он делает остальное. Музыкальные цели те же – мелодия, бас, ударные, эффекты – но модель программирования принципиально другая. Нет регистра микшера, нет расчётов периодов, нет таблиц форм огибающей. Только команды и параметры.

Для кроссплатформенных проектов подумай об абстракции звуковой системы за общим API: `sound_play_note(channel, note, instrument)`. На Spectrum поиск инструмента записывает регистры AY. На Agon – отправляет VDU-команды. Одни и те же музыкальные данные, разные бэкенды.

11.8 Практические упражнения

Упражнение 1: Обозреватель регистров. Напиши программу, позволяющую изменять любой регистр AY в реальном времени с помощью клавиатурного ввода. Отображай все 14 значений регистров на экране. Это твой единственный самый полезный инструмент отладки для работы со звуком.

Упражнение 2: Трёхканальная аранжировка. Сочини простую 16-тактовую мелодию, используя все три канала: мелодия на А, бас (buzz-bass с огибающей) на С, арпеджиированные аккорды на В. Используй тайминг на HALT из примера как отправную точку, затем перепиши его в проигрыватель на IM2.

Упражнение 4: Интеграция Vortex Tracker. Скачай Vortex Tracker II, сочини короткую мелодию, экспортируй как .pt3, и интегрируй стандартный .pt3 проигрыватель в программу для Spectrum. Убедись, что она воспроизводится корректно в эмуляторе.

AY-3-8910 на бумаге прост: 14 регистров, 3 канала, базовые формы волн. Но пропасть между «простым» и «ограниченным» заполняется техникой. Арпеджио имитируют аккорды. Злоупотребление огибающей создаёт бас. Формирование шума синтезирует ударные. Орнаменты вдыхают жизнь в статичные тоны.

А когда трёх каналов действительно мало, TurboSound удваивает их, а Next утраивает.

Итого

Архитектурный паттерн одинаков для всех конфигураций: прерывание срабатывает 50 раз в секунду, подпрограмма проигрывателя считывает данные паттернов и вычисляет значения регистров, и эти значения выгружаются в AY в тесном цикле. Пишешь ли ты свой собственный проигрыватель или интегрируешь Vortex Tracker – поток тот же. Понимание регистров означает понимание звука.

Глава 12: Цифровые барабаны и синхронизация с музыкой

“My brain is not coping with asynchronous coding well.” – Introspec, file_id.diz пати-версии Eager (to live), 3BM Open Air 2015

Демо – это не слайд-шоу эффектов. Демо – это перформанс, в котором каждое визуальное событие попадает в бит, каждый переход дышит в такт с музыкой, а зритель никогда не подозревает, что за кулисами процессор на 3,5 МГц жонглирует полудюжиной конкурирующих задач без операционной системы, без потоков и без страховочной сети.

Эта глава – об архитектуре, которая делает это жонглирование возможным. Мы потратили предыдущие главы на создание отдельных эффектов – туннелей, зумеров, скроллеров, цветовых анимаций – и в Главе 11 узнали, как чип AY производит музыку. Теперь мы должны связать всё вместе. Вопросы теперь не “как нарисовать туннель?” или “как воспроизвести ноту?”, а: как воспроизвести барабанный сэмпл, потребляющий почти весь ЦП, сохраняя плавность визуальных эффектов? Как синхронизировать смену эффектов с битом музыки? Как структурировать двухминутное демо, чтобы оно надёжно работало от начала до конца?

Ответы приходят из трёх источников. Eager Introspec’а (2015) даёт нам цифровой синтез барабанов и асинхронную генерацию кадров. GABBA diver4d (2019) показывает радикально другой подход к синхронизации с музыкой, используя видеоредактор как инструмент таймлайна. А система потоков Robus’а (2015) демонстрирует, что честная многопоточность на Z80 возможна, хотя и редко необходима.

Вместе эти три техники представляют архитектурное мышление, которое отличает коллекцию эффектов от готового демо.

12.1 Цифровые барабаны на AY

Проблема: AY не может воспроизводить сэмплы

AY-3-8910, как мы рассмотрели в Главе 11, – это PSG (Programmable Sound Generator, программируемый генератор звука). Он генерирует прямоугольные волны, шум и формы огибающей. У него нет возможности воспроизведения сэмплов, нет ЦАП, нет памяти для волновых форм. Каждый звук, который он

производит, строится из этих примитивных источников в реальном времени. Если тебе нужна реалистичная бас-бочка – с резким переходным ударом, за которым следует резонансное затухание – генератор шума и огибающей AY могут это аппроксимировать, но результат звучит безошибочно синтетически. Ему не хватает веса реального перкуссионного удара.

Но есть лазейка.

Регистры R8, R9 и R10 управляют громкостью каналов А, В и С. Каждый – это 4-битное значение (0-15). Если ты записываешь в регистр громкости раз в кадр, получаешь статический уровень громкости. Но что если записывать в него тысячи раз за кадр? Что если использовать регистр громкости как грубый 4-битный ЦАП и подавать в него последовательные значения сэмпла из оцифрованной записи?

Получаешь воспроизведение PCM. Грубое, шумное, 4-битное, но узнаваемое. AY становится проигрывателем сэмплов – не по замыслу, а грубой силой.

Цена: уничтожение ЦП

Вот проблема. Чтобы воспроизвести оцифрованный барабанный сэмпл с приемлемым качеством, нужно обновлять регистр громкости с аудио-частотой. Частота дискретизации 8 кГц означает одно обновление каждые 125 микросекунд. При 3,5 МГц 125 микросекунд – это приблизительно 437 тактов (T-state). Это плотно, но выполнимо – между записями сэмплов можно делать полезную работу.

Но 8 кГц звучит ужасно. Для бочки с ударом нужно хотя бы ощущение более высокого качества. И тут экономика рушится. При более высоких эффективных частотах дискретизации нужно прерывание или тесный цикл опроса, срабатывающий каждые 125-250 тактов (T-state). На такой частоте для чего-либо ещё почти не остаётся процессорного времени. Пока воспроизводится барабанный сэмпл, процессор – это выделенный аудио-движок. Генерация видео, скрипting, обработка ввода – всё останавливается.

Типичный сэмпл бас-бочки длится 20-40 миллисекунд для критической части атаки. При 50 Гц это 1-2 кадра. В течение этих кадров процессор занят.

Решение n1k-o: гибридный барабан

n1k-o, музыкант, создавший саундтрек Eager, нашёл решение. Ключевое наблюдение: барабанный звук имеет две различные фазы. **Атака** – начальный переходный процесс, резкий “щелчок” или “удар”, придающий бас-бочке её пробивную силу – короткая, сложная и невозможная для убедительного синтеза на AY. Но **затухание** – резонансный хвост, следующий за ним – это плавный спад громкости, именно то, что генератор огибающей AY обрабатывает естественно.

Гибридный подход: воспроизвести атаку как цифровой сэмпл (потребляя процессорное время на 1-2 кадра), затем передай управление генератору огибающей AY для затухания (потребляя ноль процессорного времени, поскольку аппаратура делает работу автоматически). Цифровая атака плюс AY-затухание равно барабанный звук с реалистичной пробивной силой сэмпла и плавным хвостом аппаратного синтеза.

На практике реализация работает так:

```

; 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, $FFFF
    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_L0
    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)

Данные сэмпла – эти 400-800 байт 4-битного PCM – берутся из реальной записи барабана, передискретизированной и квантованной до 4 бит. Переходный процесс атаки сохраняет характер оригинального инструмента: удар колотушки по мембране, начальное сжатие воздуха, резкое начало, по которому наш слух идентифицирует звук. Огибающая AY затем обеспечивает чистое, плавное затухание, которое наш слух принимает за естественный резонанс корпуса барабана.

Результат убедителен. На чипе, у которого вообще нет возможности воспроизведения сэмплов, ты слышишь нечто, похожее на настоящую бас-бочку. Не студийного качества, даже не качества Amiga, но на порядки лучше чистого AY-синтеза.

Результат убедителен. На чипе, у которого вообще нет возможности воспроизведения сэмплов, ты слышишь нечто, похожее на настоящую бас-бочку. Не студийного качества, даже не качества Amiga, но на порядки лучше чистого AY-синтеза.

Бюджет кадра: два кадра на удар

Стоимость в кадрах конкретна: два кадра на один удар барабана. В течение этих кадров приблизительно 140 000 тактов (T-state) (два полных периода кадра на Pentagon) потребляются циклом воспроизведения сэмпла. Процессор ничего другого не делает. Дисплей продолжает показывать то, что было в экранной памяти, но новые кадры не генерируются. Музыкальные данные не обрабатываются (барабан ЯВЛЯЕТСЯ музыкой на эти два кадра). Скриптовый движок не работает.

Два кадра при 50 Гц – это 40 миллисекунд. Для музыкального трека с бас-бочкой на бит при 130 BPM это примерно один удар барабана каждые 23 кадра. Два кадра из каждого 23 потрачены на воспроизведение барабана – около 9% общего процессорного времени, доставленного резкими всплесками, которые полностью монополизируют процессор.

Врезка: МСС - больше 16 уровней с одного AY

Одноканальный подход, описанный выше, даёт 16 уровней громкости (4-битный ЦАП). Но у AY три канала, и их аналоговые выходы смешиваются перед подачей на динамик. Что, если записать разные значения громкости во все три канала одновременно? Комбинированный сигнал имеет больше ступеней амплитуды, чем любой отдельный канал.

Это техника МСС (Mixed Channel Covox), документированная UnBEL!EVER'ом (Born Dead #09, 1999). Два «крайних» канала обеспечивают основной сигнал; центральный канал добавляет корректирующее значение. С предвычисленной таблицей подстановки, отображающей 8-битные значения сэмпла на три регистровых значения, можно синтезировать приблизительно 108 различных уровней амплитуды – намного лучше, чем 16.

Стандартный цикл воспроизведения МСС работает на ~24 КГц (144

такта (T-state) на сэмпл). Сверхбыстрый вариант от MOn5+Er (Born Dead #0G, 2000) использует SP как указатель данных сэмпла (POP HL читает два байта за 10 тактов) и JP (IX) для перехода цикла (8 тактов), достигая ~43,75 КГц при 80 тактах на сэмпл — приближаясь к телефонному качеству от звукового чипа, рассчитанного на прямоугольные волны.

Подвох: таблица подстановки МСС зависит от точных аналоговых уровней выхода каждого канала. AY-3-8910 и YM2149F имеют разные кривые громкости (YM ближе к линейной, AY — более логарифмическая), и даже чипы одной модели различаются между производственными партиями. Таблица на 108 уровней от UnBEL!EVER'a была рассчитана для YM2149F. На другом чипе комбинированные уровни смешиваются, и тщательно выверенные ступени амплитуды становятся неравномерными. Центральный канал вносит примерно 52% смешанного сигнала — если это соотношение дрейфует, корректирующие значения искажают сигнал, а не улучшают его.

Для демо, нацеленных на конкретную машину (скажем, Pentagon с YM2149F), МСС работает хорошо. Для кроссплатформенной совместимости одноканальный 4-битный подход безопаснее. А для экспериментов ради удовольствия — например, прогонки формул AY-beat (Приложение I) через усреднённую таблицу МСС — искажения становятся частью очарования.

Источники: UnBEL!EVER, "Воспроизведение оцифровок на AY (MCC)," Born Dead #09 (1999); MOn5+Er, Born Dead #0G (2000).

Врезка: МСС — больше 16 уровней с одного АY

Одноканальный подход, описанный выше, даёт тебе 16 уровней громкости (4-битный ЦАП). Но у AY три канала, и их аналоговые выходы смешиваются перед тем, как попасть на динамик. Что, если записать разные значения громкости во все три канала одновременно? Суммарный сигнал имеет больше ступеней амплитуды, чем любой отдельный канал.

Это техника МСС (Mixed Channel Covox), задокументированная UnBEL!EVER'ом (Born Dead #09, 1999). Два «крайних» канала обеспечивают основной сигнал; центральный канал добавляет корректирующее значение. С предвычисленной таблицей подстановки, отображающей 8-битные значения сэмпла на три регистровых значения, можно синтезировать приблизительно 108 различных уровней амплитуды — куда лучше, чем 16.

Стандартный цикл воспроизведения МСС работает на ~24 КГц (144 такта на сэмпл). Сверхбыстрый вариант от MOn5+Er (Born Dead #0G, 2000) использует SP как указатель на данные сэмпла (POP HL читает два байта за 10 тактов) и JP (IX) для ветвления цикла (8 тактов), достигая ~43,75 КГц при 80 тактах на сэмпл — приближаясь к телефонному качеству на звуковом чипе, спроектированном для прямоугольных волн.

Подвох: таблица подстановки МСС зависит от точных аналоговых

уровней выхода каждого канала. У AY-3-8910 и YM2149F разные кривые громкости (YM ближе к линейной, AY более логарифмический), и даже чипы одной модели различаются между производственными партиями. Таблица UnBEL!EVER'a на 108 уровней рассчитана для YM2149F. На другом чипе комбинированные уровни сдвигаются, и тщательно откалиброванные ступени амплитуды становятся неравномерными. Центральный канал вносит примерно 52% смешанного сигнала — если это соотношение плывёт, корректирующие значения искажают выход, а не улучшают его.

Для демо, нацеленного на конкретную машину (скажем, Pentagon с YM2149F), МСС работает хорошо. Для кроссплатформенной совместимости одноканальный 4-битный подход безопаснее. А для экспериментов ради удовольствия — например, пропустить AY-beat формулы (Приложение I) через усреднённую МСС-таблицу — искажения становятся частью шарма.

Источники: *UnBEL!EVER*, «Воспроизведение оцифровок на AY (MCC)», *Born Dead #09* (1999); *MOn5+Er*, *Born Dead #0G* (2000).

12.2 Асинхронная генерация кадров

Наивный подход не работает

Простейшая архитектура демо синхронна: сгенерируй один кадр визуального эффекта, дождись HALT (vsync), отобрази его. Сгенерируй следующий кадр, HALT, отобрази. Это то, что мы строили в каждом практическом упражнении до сих пор. Работает идеально, когда генерация кадра занимает меньше одного периода кадра.

Теперь добавим цифровые барабаны. Музыкальный движок сигнализирует: “воспроизведи бочку на следующий бит”. Подпрограмма воспроизведения сэмпла захватывает ЦП на два кадра. В течение этих двух кадров новые видеокадры не генерируются. Когда барабан заканчивается, дисплей показывал один и тот же кадр три обновления подряд (последний сгенерированный кадр был показан раз нормально, затем дважды во время удара барабана). Визуальный эффект заикается.

С одним ударом барабана каждые 23 кадра зритель видит короткое замирание каждые полсекунды. Это заметно. Это некрасиво. Это неприемлемо для конкурсного демо.

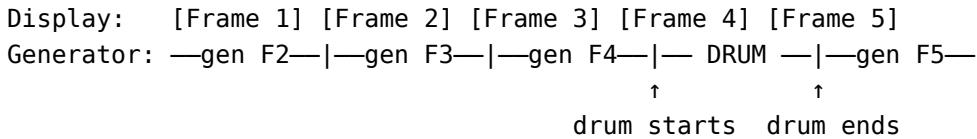
```
Display: [Frame 1] [Frame 2] [Frame 3] [Frame 4] [Frame 5]
Generator: —gen F2—|—gen F3—|—gen F4—|— DRUM —|—gen F5—
                           ↑           ↑
                           drum starts   drum ends
```

During the drum hit, the display shows Frame 4 (already generated).
Frame 5 generation resumes immediately after the drum finishes.

Архитектура Introspec'a в Eager разделяет генерацию кадров и их отображение. Визуальный движок не генерирует один кадр и сразу его показывает. Вместо

этого он генерирует кадры в буфер – столько, сколько влезет – а система отображения показывает их с постоянной частотой 50 Гц независимо от того, что делает генератор.

Механизм – это атрибутные кадры с двойной буферизацией. Две страницы атрибутных данных существуют в памяти. Пока одна страница отображается (ULA читает из неё во время обновления экрана), генератор записывает следующий кадр в другую страницу. Когда новый кадр готов, движок переключает страницы: только что сгенерированный кадр становится страницей экрана, а старая страница экрана становится новой целью генерации.



During the drum hit, the display shows Frame 4 (already generated).
Frame 5 generation resumes immediately after the drum finishes.

Но простая двойная буферизация даёт только один кадр запаса. Если барабан потребляет два кадра, нужно сгенерировать два кадра вперёд. Вот где асинхронная генерация Introspec'а по-настоящему расходится с простой двойной буферизацией: движок может **накапливать** несколько кадров впрок.

На 128K Spectrum переключение банков памяти обеспечивает пространство. Атрибутные кадры маленькие – 768 байт каждый. Одна 16-килобайтная страница памяти может вместить примерно 20 атрибутных кадров. Генератор работает так быстро, как может, записывая кадр за кадром в буфер. Система отображения читает из буфера с постоянной частотой 50 Гц. Когда генератор быстрее реального времени (что обычно так, поскольку атрибутная плазма дешёвая), буфер наполняется. Когда удар барабана приостанавливает генерацию, система отображения расходует буфер. Пока буфер не опустеет, зритель видит плавную анимацию на 50 Гц.

Динамика буфера

Думай об этом как о задаче производитель-потребитель, но на машине без параллелизма.

Производитель – генератор эффекта плазмы/туннеля/зумера. Он производит атрибутные кадры с переменной частотой – иногда быстрее 50 Гц (когда расчёт прост и барабаны не воспроизводятся), иногда ноль (во время воспроизведения барабана).

Потребитель – система отображения, читающая один кадр за обновление экрана ровно при 50 Гц.

Буфер сидит между ними, поглощая разницу.

Динамика проста:

Критическое ограничение: **буфер никогда не должен опустеть во время удара барабана**. Если два удара происходят в быстрой последовательности

– скажем, бочка-малый с разрывом в два кадра – буферу нужен запас минимум в четыре кадра. Скриптовый движок Introspec'а управляет этим, зная музыкальный таймлайн заранее. Когда приближается плотный барабанный пассаж, движок генерирует дополнительные кадры для наполнения буфера. Когда следует тихий фрагмент, буфер естественным образом заполняется.

Подвох: если барабанный паттерн слишком плотный – слишком много ударов слишком близко друг к другу – генератор не успевает. Буфер пустеет, и дисплей повторяет кадр. Это жёсткое ограничение архитектуры, и оно повлияло на композицию n1k-o. Музыка была написана со знанием ёмкости движка: удары барабанов расположены достаточно далеко друг от друга, чтобы генератор всегда мог восстановиться. Музыкант и кодер проектировали вместе, каждый понимая ограничения другого.

12.3 Скриптовый движок

Зачем нужен скрипт

К этому моменту список вещей, требующих координации, длинный:

- Генератор визуального эффекта (какой эффект активен, какие параметры он использует)
- Музыкальный проигрыватель (какой паттерн воспроизводится, когда срабатывают барабаны)
- Буфер кадров (насколько он полон, когда генерировать больше)
- Переходы между эффектами (плавное затухание одного, плавное появление следующего)
- Общий таймлайн (демо длится две минуты – что происходит когда)

```
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
```

Внешний скрипт: последовательность эффектов

Внешний скрипт – это линейная последовательность команд, управляющих общей структурой демо. Думай о нём как о сет-листе для концерта:

```
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

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
```

Внутренний скрипт: вариации внутри эффекта

Внутри одного эффекта параметры меняются со временем. Частоты плазмы туннеля смещаются, цветовая палитра вращается, скорость зума ускоряется. Эти вариации контролируются внутренним скриптом – последовательностью изменений параметров для конкретного эффекта, привязанных к номерам кадров:

```
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
```

Внутренний скрипт работает независимо от внешнего. Когда внешний скрипт говорит “запусти туннель на 200 кадров”, внутренний скрипт управляет визуальной эволюцией в рамках этих 200 кадров.

kWORK: ключевая команда

Самая важная команда в системе скриптинга – то, что Introspec называет **kWORK**: “сгенерируй N кадров, затем показывай их независимо от генерации”. Эта единственная команда – мост между системой скриптинга и асинхронной архитектурой.

Когда движок встречает kWORK 8, он:

```
; 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
```

Это разделение – генерируй сейчас, показывай потом – фундаментальная основа для асинхронной работы. Без kWORK движок был бы заперт в синхронном цикле генерация-отображение-генерация-отображение без запаса для прерываний барабанами.

```

; 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

```

В 2019 году diver4d (из 4D+TBK) занял первое место на CAFE с GABBA, габбер-тематическим демо с беспощадно точной аудиовизуальной синхронизацией. Синхронизация была настолько точной, что каждый визуальный акцент попадал точно на музыкальный бит, каждый переход совпадал с границей фразы, и вся постановка ощущалась как музыкальный клип, а не демо.

12.4 Инновация GABBA: видеоредактор как инструмент таймлайна

В 2019 году diver4d (из 4D+TBK) занял первое место на CAFE с GABBA, габбер-тематическим демо с беспощадно точной аудиовизуальной синхронизацией. Синхронизация была настолько точной, что каждый визуальный акцент попадал точно на музыкальный бит, каждый переход совпадал с границей фразы, и вся постановка ощущалась как музыкальный клип, а не демо.

Технический сюрприз был в рабочем процессе, а не в коде.

Проблема код-ориентированной синхронизации

Традиционный подход к синхронизации с музыкой в ZX-демо – встраивание данных тайминга в код. Ты знаешь, что бочка ударяет на кадре 47, и пишешь команду скрипта, запускающую визуальное событие на кадре 47. Затем смотришь демо, решаешь, что тайминг слегка не совпадает, меняешь число на 49, перекомпилируешь, перетестируешь и повторяешь. Для двухминутного демо при 50 fps это 6 000 кадров потенциальных точек синхронизации. Подгонка их всех методом проб и ошибок занимает недели.

Eager Introspec'а был построен именно так, и разработка была изнурительной. Каждая поправка синхронизации требовала перекомпиляции – ассемблирования Z80-кода, загрузки бинарника в эмулятор, просмотра соответствующего

фрагмента, записи того, что не так, редактирования исходника и повторения. Цикл обратной связи измерялся минутами на итерацию.

Ответ diver4d: Luma Fusion

diver4d полностью обошёл цикл код-редактирование-компиляция-тест. Он использовал **Luma Fusion**, видеоредактор для iOS, как инструмент синхронизации.

Рабочий процесс:

Идея проста: используй правильный инструмент для работы. Видеоредактор специально создан для покадровой мультимедийной синхронизации. Ассемблер Z80 – нет. Выполняя творческую работу по синхронизации в редакторе, а реализацию – в ассемблере, diver4d разделил художественные решения и инженерные ограничения.

2. **diver4d записал каждый визуальный эффект**, работающий при 50 fps в эмуляторе, и экспортировал записи как видеоклипы.
 3. **В Luma Fusion** он расположил видеоклипы на таймлайне 50 fps рядом с аудиотреком. Он мог прокручивать демо покадрово, видя точно, как каждый визуальный элемент совпадает с каждым музыкальным событием. Перемещение перехода было так же просто, как перетаскивание клипа на таймлайне.
 4. **Когда тайминг был правильным в редакторе**, он извлекал номера кадров для каждого перехода и смены эффекта и записывал эти числа в данные скрипта Z80.
-

Что это меняет

Непосредственная выгода – скорость. Подгонка тайминга синхронизации в видеоредакторе занимает секунды. Подгонка в ассемблере – минуты. На сотнях точек синхронизации кумулятивная экономия времени огромна. Но более глубокая выгода – творческая свобода. Когда итерация дешёвая, ты больше экспериментируешь. Ты пробуешь переход на два кадра раньше, смотришь, как ощущается, пробуешь на два кадра позже. Ты замечаешь, что визуальный эффект лучше работает, попадая чуть до бита (техника, заимствованная из кинообработки, где монтажные склейки на бит ощущаются запоздалыми из-за времени реакции человека). Ты никогда бы не обнаружил эту идею через код-ориентированную итерацию – цикл обратной связи слишком медленный.

Ограничение в том, что этот рабочий процесс лучше всего работает для демо с фиксированным таймингом – где демо всегда воспроизводится одинаково. Если нужны интерактивные или генеративные элементы, реагирующие на условия во время выполнения, нужен код-ориентированный подход. Но для подавляющего большинства ZX-демо, которые являются линейными постановками с фиксированным таймлайном, рабочий процесс через видеоредактор превосходит.

GABBA продемонстрировала, что инструменты производства демосцены не обязаны быть ретро. Z80-код из 1985 года. Рабочий процесс синхронизации

может быть из 2019-го. Противоречия нет.

12.5 Потоки на Z80: другой путь

Robus, публикуясь в Нуре в 2015 году, представил технику, атакующую проблему параллелизма с совершенно другого угла: настоящая многопоточность на Z80.

Проблема, переформулированная

```
; 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
```

Что если Z80 мог бы выполнять две задачи одновременно?

Переключение контекста на базе IM2

Он может, до некоторой степени. Прерывание IM2 на Z80 обеспечивает естественную точку переключения контекста. Каждый кадр срабатывает прерывание. Если обработчик прерывания сохраняет состояние текущей задачи и загружает состояние другой задачи, ты получаешь вытесняющую многопоточность.

Процедура SwitchThread Robus'a делает именно это:

```
; 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

```

Каждый поток получает собственный **128-байтный стек и выделенную страницу памяти** (один из восьми 16-килобайтных банков 128K Spectrum). Стек маленький, но достаточный – Z80-код редко имеет глубокую вложенность. Выделенная страница памяти даёт каждому потоку собственное рабочее пространство без интерференции с другим.

Как это работает на практике

В демо WAYHACK Robus'a два потока работают параллельно:

```

Frame 2: Interrupt → save Thread 1 → restore Thread 2 → Thread 2 runs
Frame 3: Interrupt → save Thread 2 → restore Thread 1 → Thread 1 runs
...

```

Ни один поток не знает о другом. Каждый работает в собственной странице памяти с собственным стеком. Каждый кадр срабатывает прерывание IM2, и SwitchThread переключается между ними. Поток 1 получает один кадр процессорного времени, затем Поток 2 получает один кадр, и так далее.

Результат: текстовый скроллер работает на стабильных 25 Гц (каждый второй кадр), и визуальный эффект работает на 25 Гц. Ни одна задача не должна знать о существовании другой. Никакого кооперативного планирования, никаких точек уступки, никакого ручного чередования. Прерывание обрабатывает всё.

Модель потоков

Модель проста:

```

Frame 2: Interrupt → save Thread 1 → restore Thread 2 → Thread 2 runs
Frame 3: Interrupt → save Thread 2 → restore Thread 1 → Thread 1 runs
...

```

Практические соображения

Собственная оценка Robus'a характерно честна: **“Честная многопоточность редко требует больше двух потоков”** на Z80. Накладные расходы на переключение контекста (сохранение и восстановление SP плюс переключение

страницы памяти) скромны – возможно, 100 тактов (T-state) – но каждый дополнительный поток вдвое уменьшает доступное процессорное время на поток. С двумя потоками каждый получает 25 Гц. С тремя – каждый получает примерно 16,7 Гц. На машине, где визуальная плавность требует близко к 50 Гц, два потока – практический предел.

Подход потоков ортогонален подходу асинхронной буферизации Introspec'a. Их можно комбинировать: один поток генерирует кадры эффекта в буфер, другой обрабатывает музыку и воспроизведение барабанов. На практике такая комбинация редка – две техники решают одну и ту же проблему (чередование ресурсоёмких задач) через разные механизмы, и большинство демо-кодеров выбирают одну или другую исходя из конкретных требований их постановки.

Потоки лучше всего работают, когда две задачи по-настоящему независимы и ни одна не требует больше 25 Гц. Подход с асинхронным буфером лучше всего работает, когда одна задача (визуальные эффекты) требует 50 Гц, а другая (барабаны) – непредсказуемые всплески. Для архитектуры Eager, где визуальная плавность была первостепенна, а тайминг барабанов диктовался музыкой, победил подход с буфером. Для архитектуры WAYHACK, где две устойчивые задачи работали параллельно, победили потоки.

\$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)

Давай построим минимальный движок демо, связывающий вместе концепции из этой главы. Цель – не уровень сложности Eager – это каркас, демонстрирующий архитектуру.

```
; 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
```

- **Три простых эффекта:** плазма (на атрибутах, из Главы 9), цветные полосы (горизонтальные атрибутные полосы) и текстовый скроллер.

- **AY-музыка**, воспроизводимая через прерывание IM2 (с использованием .pt3 проигрывателя, как описано в Главе 11).
- **Цифровой сэмпл бас-бочки**, воспроизводимый на бит, крадущий 2 кадра ЦП.
- **Простой скрипт таймлайна**, переключающий эффекты в определённых точках.
- **Атрибуты с двойной буферизацией** для поглощения пауз от ударов барабанов.

```

; 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

$8000-$FFFF 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)

; 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)

; 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

```

```

; 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

; 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

```

Обработчик прерываний дисплея

```

; 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)

```

Диспетчер генератора эффектов

```

; 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

```

Наблюдения

Этот каркас намеренно прост. Продакшн-движок добавил бы:

- **Внутренние скрипты** для вариации параметров внутри каждого эффекта.
- **Эффекты переходов** (кроссфейды между двумя атрибутными буферами).
- **Несколько звуков барабанов** (бочка, малый, хай-хэт), каждый со своими данными сэмпла.
- **Мониторинг уровня буфера**, чтобы генератор мог приоритизировать восстановление после плотных барабанных пассажей.
- **Переключение банков памяти** для хранения большего количества кадров и поддержки более объёмных данных эффектов.

Но даже в минимальной форме эта архитектура демонстрирует ключевые принципы:

-
2. **Удары барабанов поглощаются буфером.** Когда `play_kick_drum` потребляет два кадра, обработчик прерываний дисплея продолжает показывать буферизованные кадры. Зритель не видит заикания.
 3. **Скрипт управляет таймлайном.** Добавление нового эффекта или изменение последовательности означает редактирование таблицы данных `timeline`, а не перестройку кода движка.
 4. **Музыкальный проигрыватель работает в обработчике прерываний.** Он обновляет регистры AY каждый кадр независимо от того, что делает основной цикл. Единственное взаимодействие – флаг `drum_pending` – однобайтный “почтовый ящик” между обработчиком прерываний и основным циклом.

Это архитектура демо. Не эффекты, не музыка, не арт – *сантехника*, которая заставляет всё это работать вместе. Это наименее видимая часть демо и самая сложная в реализации. Introspec потратил десять недель на Eager, и архитектура заняла больше времени, чем любой отдельный эффект.

12.7 Практические упражнения

Упражнение 2: Добавь барабан. Запиши (или синтезируй) 4-битный сэмпл бас-бочки (400-800 байт). Добавь подпрограмму `play_kick_drum` и запускай её каждые 25 кадров. Убедись, что дисплей остаётся плавным во время воспроизведения барабана. Какова максимальная частота ударов, прежде чем буфер опустеет?

Упражнение 3: Многоэффектный таймлайн. Добавь второй эффект (цветные полосы или текстовый скроллер). Напиши скрипт таймлайна, переключающий эффекты каждые 3-4 секунды. Убедись, что переходы происходят в правильный кадр.

Упражнение 4: Синхронизация с музыкой. Загрузи короткую .pt3 мелодию и модифицируй проигрыватель, чтобы он устанавливал `music_drum_flag`, когда происходит определённое событие паттерна (например, нота на канале C ниже определённой высоты). Теперь барабаны управляются музыкой, а не фиксированным счётчиком кадров. Это настоящая синхронизация с музыкой.

Упражнение 5: Рабочий процесс через видеоредактор. Запиши работающее демо в эмуляторе при 50 fps. Импортируй запись в видеоредактор (любой, поддерживающий покадровое редактирование). Подгони номера кадров скрипта таймлайна на основе того, что видишь в редакторе. Почувствуй разницу в скорости итерации по сравнению с синхронизацией через код.

Итого

Эта глава была не об отдельном эффекте или технике. Она была об архитектуре – невидимой структуре, которая позволяет демо существовать как связная, синхронизированная, двухминутная постановка, а не как коллекция разрозненных экранов.

Рассмотренные решения дополняют друг друга:

Глава 13: Мастерство sizecoding

"It was like playing puzzle-like games – constant reshuffling of code to find shorter encodings." – UriS, о написании NHBF (2025)

Существует категория соревнований демосцены, где ограничение – не время, а *пространство*. Вся твоя программа – код, рисующий экран, производящий звук, обрабатывающий покадровый цикл, хранящий необходимые данные – должна уместиться в 256 байт. Или 512. Или 1К, или 4К, или 8К. Ни байтом больше. Файл измеряется, и если он 257 байт, он дисквалифицируется.

Это **sizecoding**-соревнования, и они производят одни из самых выдающихся работ на ZX Spectrum-сцене. 256-байтное интро, заполняющее экран анимированными паттернами и воспроизводящее узнаваемую мелодию – это форма сжатия настолько экстремальная, что в ней трудно поверить, пока не прочитаешь код. Разрыв между тем, что видит зритель, и размером файла, который это производит – этот разрыв и есть искусство.

Эта глава о мышлении, техниках и конкретных трюках, которые делают sizecoding возможным.

13.1 Что такое sizecoding?

Демо-соревнования обычно предлагают несколько категорий с ограничением по размеру:

Категория	Лимит	Что влезает
256 байт	256	Один плотный эффект, может быть простой звук
512 байт	512	Эффект с базовой музыкой или два простых эффекта
1К интро	1 024	Несколько эффектов, полноценная музыка, переходы
4К интро	4 096	Короткое демо с несколькими частями
8К интро	8 192	Полированное мини-демо

Лимиты абсолютны. Файл измеряется в байтах, и торговли нет.

Что делает sizecoding захватывающим – оно инвертирует обычную иерархию оптимизации. В мире демосценических эффектов с подсчётом тактов ты оптимизируешь на *скорость* – развортываешь циклы, дублируешь данные, генерируешь код, всё обменивая пространство на время. Sizecoding инвертирует это.

Скорость не важна. Читаемость не важна. Единственный вопрос: можно ли сделать это на один байт короче?

UriS, написавший 256-байтное интро NHBF для Chaos Constructions 2025, описал процесс как “playing puzzle-like games”. Описание точное. Sizecoding – это головоломка, где фигуры – инструкции Z80, поле – 256 байт ОЗУ, а лучшие решения включают ходы, решающие несколько задач одновременно.

Сдвиг мышления:

- **Каждый байт драгоценен.** 3-байтная инструкция там, где достаточно 2-байтной – это 0,4% всей программы. На 256 байт один сэкономленный байт – это как сэкономить 250 байт в 64К-программе.
- **Код и данные перекрываются.** Одни и те же байты, выполняемые как инструкции, могут служить данными. Z80 не видит разницы – только путь счётчика команд через память различает код и данные.
- **Выбор инструкций диктуется размером, не скоростью.** RST \$10 стоит 1 байт. CALL \$0010 делает то же самое за 3 байта. В обычном демо ты бы и не заметил. В 256 байтах эти 2 байта – разница между наличием звука и его отсутствием.
- **Начальное состояние – бесплатные данные.** После загрузки регистры имеют известные значения. Память по определённым адресам содержит известные данные. Sizecoding-мастер использует каждый бит этого бесплатного состояния.
- **Самомодифицирующийся код (SMC) – не трюк, а необходимость.** Когда ты не можешь позволить себе отдельную переменную, ты модифицируешь операнд инструкции на месте.

Инструментарий сайзкодера на Z80

Некоторые трюки встречаются в размерно-оптимизированных интро настолько часто, что образуют общий словарь. Знать их наизусть – их стоимость в байтах, их побочные эффекты – это необходимое условие для серьёзного sizecoding.

Предположения об инициализации регистров. Когда программа на Spectrum запускается из BASIC (через RANDOMIZE USR), состояние процессора не случайно. После CLEAR и перед вызовом USR A обычно равен 0, BC содержит адрес USR, DE и HL имеют известные значения из интерпретатора BASIC, указатель стека установлен на адрес CLEAR, и прерывания разрешены. Многие из этих значений достаточно стабильны, чтобы на них полагаться. Если твоей программе нужен A = 0 в начале, не пиши XOR A – он уже ноль. Если тебе нужен 16-битный счётчик, начинающийся с 0, проверь, не содержит ли уже DE или HL ноль или полезное значение. Один байт сэкономлен здесь, два байта там – они складываются в разницу между 260 байтами и 256.

Область системных переменных (\$5C00-\$5CB5) – ещё один источник бесплатных данных. Интерпретатор BASIC поддерживает более 100 байт состояния по известным адресам. Если тебе нужно значение 2, ты можешь найти его по адресу, где хранится номер текущего потока. Если нужен \$FF, несколько полей системных переменных содержат его. Чтение с фиксированного адреса стоит 3 байта (LD A, (nn)), но если оно заменяет 2-байтную загрузку плюс некоторые вычисления, ты выигрываешь.

DJNZ как короткий обратный переход. DJNZ label занимает 2 байта, столько же, сколько JR label – но при этом ещё и декрементирует В. Если В ненулевой и тебе нужен обратный переход, DJNZ делает и то, и другое бесплатно. Даже когда тебе не нужен декремент В, DJNZ всё равно 2 байта, та же стоимость, что и JR. Но если В достигает нуля в точности в тот момент, когда ты хочешь провалиться дальше, ты объединил счётчик цикла и переход в одну инструкцию. Сайзкодеры регулярно строят циклы так, чтобы естественный обратный отсчёт В совпадал с условием выхода.

RST как 1-байтный CALL. Z80 резервирует восемь адресов рестарта: \$00, \$08, \$10, \$18, \$20, \$28, \$30, \$38. RST н помещает адрес возврата в стек и прыгает по целевому адресу – то же самое, что CALL н – но в 1 байт вместо 3. На Spectrum ROM размещает полезные подпрограммы по некоторым из этих адресов:

- RST \$10 – печать символа (подпрограмма ROM по адресу \$0010)
- RST \$20 – получение следующего символа из BASIC (менее полезно для демо)
- RST \$28 – вход в калькулятор с плавающей точкой (полезно для математики)
- RST \$38 – обработчик маскируемого прерывания (IM 1 прыгает сюда)

В обычном демо эти подпрограммы ROM слишком медленны для вызова в плотном цикле. В 256-байтном интро экономия 2 байт на вызов стоит потери скорости. Если твоя программа вызывает RST \$10 шесть раз для печати символов, это 12 байт экономии по сравнению с шестью инструкциями CALL \$0010. Двенадцать байт – это почти 5% от 256.

Перекрывающиеся инструкции. Z80 декодирует инструкции побайтно, без требований к выравниванию. Если ты прыгнешь в середину многобайтовой инструкции, процессор начнёт декодирование заново с этой точки. Это значит, что ты можешь спрятать одну инструкцию внутри другой:

```
ld a, $AF          ; opcode $3E, operand $AF
                     ; BUT: $AF is XOR A
```

Если процессор выполняет код с начала, он видит LD A, \$AF (2 байта). Если другой путь исполнения прыгает на второй байт, он видит XOR A (1 байт). Один байт служит двум целям. Техника хрупкая – она требует идеального контроля над всеми путями исполнения – но в соревновательном коде хрупкость допустима.

Распространённый паттерн: байт \$18 – это JR d (относительный переход). Если тебе нужно значение \$18 как данные и нужен переход в этом месте, один и тот же байт делает и то, и другое. Операнд, следующий за ним, одновременно является смещением перехода и (с другой точки зрения) следующим фрагментом данных.

Эксплуатация состояния флагов. Каждая арифметическая и логическая инструкция устанавливает флаги. Сайзкодеры запоминают, какие флаги затрагивает каждая инструкция, и используют результаты вместо того, чтобы вычислять их отдельно. После DEC В флаг нуля сообщает, достиг ли В нуля – никакого CP 0 не нужно. После ADD A, n флаг переноса сообщает, вышел ли результат за 255. После AND mask флаг нуля сообщает, были ли установлены какие-либо замаскированные биты.

Самый глубокий трюк с флагами – SBC A, A: если перенос установлен, А становится \$FF; если перенос сброшен, А становится \$00. Один байт, без ветвления,

полная битовая маска из флага. Сравни с альтернативой через ветвление:

```
; 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
```

Пять байт сэкономлено. В 256-байтном интро это два процента от всей программы.

13.2 Анатомия 256-байтного интро: NHBF

NHBF (No Heart Beats Forever) создано UriS для Chaos Constructions 2025, вдохновлённое RED REDUX с Multimatograf 2025. Оно производит текст с экранными эффектами и музыку – зацикленные пауэр-аккорды прямоугольной волной со случайными мелодическими нотами пентатоники – всё в 256 байтах.

Музыка

При 256 байтах ты не можешь включить проигрыватель трекера или таблицу нот. NHBF управляет чипом AY напрямую. Пауэр-аккорды захардкодены как непосредственные значения в инструкциях записи регистров AY – те же байты, что формируют операнд LD A, n, являются музыкальной нотой. Канал мелодии использует псевдослучайный генератор (обычно LD A, R – чтение регистра обновления – с последующим AND для маскирования диапазона) для выбора из пентатонической шкалы. Пентатоническая шкала звучит приятно независимо от того, какие ноты оказываются рядом, поэтому мелодия звучит намеренно, хотя она случайна. Два байта на “случайное” число; пять нот, которые никогда не конфликтуют.

Визуальная часть

Печать текста через ПЗУ – RST \$10 выводит символ за 1 байт на вызов – самый дешёвый способ получить пиксели на экране. Но даже 20-символьная строка стоит 40 байт (коды символов + вызовы RST). Sizecoding-мастера ищут способы сжать ещё: перекрытие строковых данных с другим кодом или вычисление символов по формуле.

Головоломка: поиск перекрытий

UriS описывает основной процесс как постоянное перестановку. Ты пишешь первую версию на 300 байт, затем вглядываешься в неё. Ты замечаешь, что счётчик цикла визуального эффекта в итоге содержит значение, нужное тебе как номер регистра AY. Убери LD A, 7, который бы его установил – цикл и

так оставил 7 в А. Два байта сэкономлены. Подпрограмма очистки экрана использует LDIR, который обнуляет ВС. Расположи код так, чтобы следующему фрагменту нужен ВС = 0, и сэкономь LD BC, 0 – ещё 3 байта.

Каждая инструкция производит побочные эффекты – значения регистров, состояния флагов, содержимое памяти – и искусство заключается в расположении инструкций так, чтобы побочные эффекты одной подпрограммы были входами другой.

Открытие Art-Top

Во время разработки Art-Top заметил нечто замечательное: значения регистров, оставшиеся от подпрограммы очистки экрана, случайно совпали с точной длиной, нужной для текстовой строки. Не запланировано. UriS написал очистку экрана, затем вывод текста, и два фрагмента случайно разделили состояние регистра, что устранило отдельный счётчик длины.

Такое случайное совпадение перекрытий – сердцевина 256-байтного кодинга. Ты не можешь это запланировать. Ты можешь только создать условия, при которых это может произойти, постоянно переставляя код и наблюдая за случайными совпадениями. Когда находишь одно – это как обнаружить, что два кусочка пазла из разных головоломок идеально подходят друг к другу.

Байтовый бюджет

При работе на 256 байт примерный бюджет помогает спланировать до написания хотя бы одной инструкции. Вот реалистичная раскладка для типичного 256-байтного интро для ZX Spectrum с визуальной частью и звуком:

Компонент	Байты	Примечания
Заливка пикселей (дизеринг/очистка)	18-25	LD HL, LDIR или компактный цикл заполнения
Инициализация AY	16-22	Микшер, громкость, начальный тон – через запись в порты
Синхронизация с кадром в главном цикле	1	HALT
Обновление тона AY за кадр	10-14	Выбор регистра, запись периода тона
Ядро визуального эффекта	30-50	Внутренний цикл, вычисляющий и записывающий атрибуты
Внешний цикл / управление строками	8-12	Счётчик строк, счётчик столбцов, переходы
Обновление счётчика кадров (SMC)	6-8	Чтение, инкремент, запись обратно в инструкцию
Возврат к началу главного цикла	2	JR main_loop
Итого каркас	~91-134	До любого кода, специфичного для эффекта

Остаётся 122-165 байт на собственно творческое содержание – визуальную формулу, таблицы данных, дополнительную звуковую логику, текстовые строки или что угодно ещё, что делает интро *твоим*. Каркас стоит дорого. Вот почему сайзкодеры так отчаянно борются за каждый байт в обвязке: каждый байт, сэкономленный в каркасе – это байт, выигранный для искусства.

Посмотри на сопровождающий пример `intro256.asm`. Его цикл заливки пикселей занимает 18 байт. Настройка AY – 20 байт. Каркас главного цикла (HALT, чтение счётчика кадров, обновление бордюра) – 8 байт. Обновление тона AY – 13 байт. Визуальный эффект – паттерн интерференции Муара, вычисленный исключительно из регистровой арифметики – занимает 36 байт. Запись счётчика кадров и переход на начало цикла – 8 байт. Итого: около 103 байт каркаса и 36 байт эффекта. Это соотношение – примерно 3:1 каркас к эффекту – типично. Чем лучше ты сожмёшь каркас, тем больше места останется для творческого самовыражения.

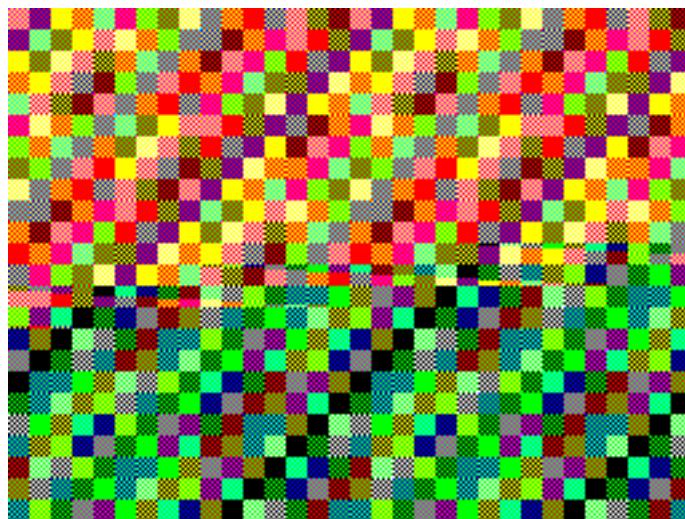


Рис. 29: Вывод 256-байтного интро – анимированный паттерн интерференции Муара с цветовым циклированием, сгенерированный исключительно из регистровой арифметики

Ключевые техники на 256 байтах

1. Используй начальное состояние регистров и памяти. После стандартной загрузки с ленты регистры содержат известные значения: A часто содержит последний загруженный байт, BC – длину блока, HL указывает proximity конца загруженных данных. Область системных переменных (\$5C00-\$5CB5) содержит известные значения. Экранная память чиста после CLS. Каждое известное значение, которое ты используешь вместо явной загрузки, экономит 1-3 байта.

2. Перекрывай код и данные. Байт \$3E – опкод для LD A, n и также значение 62 – ASCII-символ, координата экрана или значение регистра AY. Если твоя программа выполняет этот байт как инструкцию и читает его как данные из другого пути кода, ты заставил один байт делать две работы. Типичный паттерн: непосредственный операнд LD A, n одновременно служит данными, которые другая подпрограмма читает через LD A, (addr), указывая на `instruction_address + 1`.

3. Выбирай инструкции по размеру.

Большая кодировка	Маленькая кодировка	Экономия
CALL \$0010 (3 байта)	RST \$10 (1 байт)	2 байта
JP label (3 байта)	JR label (2 байта)	1 байт
LD A, 0 (2 байта)	XOR A (1 байт)	1 байт
CP 0 (2 байта)	OR A (1 байт)	1 байт

Инструкции RST критичны. RST n – это 1-байтный CALL к одному из восьми адресов (\$00, \$08, \$10, \$18, \$20, \$28, \$30, \$38). На Spectrum RST \$10 вызывает вывод символа через ПЗУ, RST \$28 входит в калькулятор. В обычном демо эти подпрограммы ПЗУ слишком медленны. На 256 байтах экономия 2 байт на CALL – это всё.

Каждый JP в 256-байтном интро должен быть JR – вся программа помещается в диапазон -128..+127.

4. Самомодифицирующийся код (SMC) для повторного использования последовательностей. Нужна подпрограмма, оперирующая двумя разными адресами? Захардкодь первый и запатчь операнд для второго вызова. Дешевле, чем передача параметров.

5. Математические соотношения между константами. Если твоей музике нужен период тона 200 и эффекту нужен счётчик цикла 200, используй один и тот же регистр. Если одно значение вдвое больше другого, используй ADD A, A (1 байт) вместо загрузки второй константы (2 байта).

13.3 Знаменитые 256-байтные интро: что делало их гениальными

Категория 256-байтных интров для ZX Spectrum имеет богатую историю. Изучение победивших работ показывает, какие эффекты помещаются в 256 байт и какие творческие стратегии приводят к успеху.

Атрибутные эффекты доминируют. Причина арифметическая: область атрибутов Spectrum – это 768 байт (32 x 24), и её можно заполнить вычислением паттерном, используя плотный вложенный цикл из 15-20 байт. Пиксельные эффекты требуют адресации 6144 байт чересстрочной экранной памяти – значительно больше кода только на вычисление адреса. На 256 байтах ты просто не можешь позволить себе эти накладные расходы. Поэтому подавляющее большинство 256-байтных интров работают в пространстве атрибутов: цветовые плазмы, интерференционные паттерны, градиентные анимации, циклирование цветов. Пиксельная память либо остаётся пустой, либо получает одноразовую заливку дизерингом, либо остаётся с тем, что туда записал ROM.

Генеративный звук побеждает секвенсированный. Таблица нот для мелодии стоит байтов – даже простая 8-нотная последовательность это 8 байт плюс логика индексации. На 256 байтах выигрышная стратегия – выводить звук из состояния эффекта. Используй счётчик кадров как период тона (высота непрерывно меняется). Используй байт из визуального вычисления как параметр

шума. Или используй LD A, R – чтение регистра обновления Z80, который инкрементируется при каждом считывании инструкции – как псевдослучайный источник, а затем замаскируй его до пентатонического диапазона. Звук не будет композицией, но он будет *присутствовать*, и зрители запомнят “то крохотное интро, в котором была музыка”.

ROM - твоя библиотека. Каждый байт 16-килобайтного ROM Spectrum доступен и не считается в твой лимит размера. RST \$10 печатает символы, используя полный шрифтовой рендеринг ROM – 96 печатных символов, 8x8 пикселей каждый, с управлением курсором. Это тысячи байт кода рендеринга, доступных за 1 байт на вызов. RST \$28 открывает доступ к калькулятору с плавающей точкой, который умеет вычислять синус, косинус и квадратные корни – операции, которые стоили бы десятков байт для самостоятельной реализации. Цена – скорость (подпрограммы ROM медленные), но в 256-байтном интро, работающем на 50fps с простым эффектом, у тебя часто есть такты в запасе.

Побеждают те работы, которые выглядят невозможными при своём размере. Судьи и зрители реагируют на разрыв между воспринимаемой сложностью и размером файла. 256-байтное интро с плавной цветовой плазмой и узнаваемой мелодией вызывает больше аплодисментов, чем интро с чуть лучшей визуальной частью, но без звука. Трюк в том, чтобы выбрать эффект, который *выглядит* сложным, но *кодируется* дёшево. Интерференционные паттерны на основе XOR идеальны: визуально замысловатые, математически тривиальные. Циклирование цветов через атрибуты – ещё один: глаз воспринимает движение и глубину, но код – это просто инкремент значений в цикле. Диагональные скроллинговые паттерны, шахматные анимации, расширяющиеся кольца – всё это можно произвести менее чем из 20 байт кода внутреннего цикла, если формула выбрана аккуратно.

13.4 Трюк LPRINT

В 2015 году diver4d опубликовал “Secrets of LPRINT” на Нуре, документируя технику старше самой демосцены – впервые появившуюся в пиратских загрузчиках кассетного софта в 1980-х.

Как это работает

Системная переменная по адресу 23681 (\$5C81) управляет тем, куда подпрограммы вывода BASIC направляют данные. Обычно она указывает на буфер принтера. Измени её, чтобы она указывала на экранную память, и LPRINT пишет прямо на экран:

Этот единственный POKE перенаправляет канал принтера на \$4000 – начало экранной памяти.

Эффект транспозиции

Визуальный результат – не просто текст на экране – это *транспонированный* текст. Экранная память Spectrum чересстрочная (Глава 2), но драйвер прин-

тера пишет последовательно. Данные попадают в экранную память согласно линейной логике драйвера, но отображаются согласно чересстрочной раскладке. Результат проходит через 8 визуальных состояний по мере продвижения через трети экрана – каскад данных, который строится горизонтальными полосами, смещаясь и рекомбинируя.

С другими символьными данными – графическими символами, UDG или тщательно подобранными ASCII-последовательностями – транспозиция производит поразительные визуальные паттерны. Оператор LPRINT обрабатывает всю адресацию экрана, рендеринг символов и продвижение курсора. Твоя программа предоставляет только данные.

От пиратских загрузчиков к демо-арту

diver4d проследил трюк до пиратских загрузчиков с кассет. Пираты, добавлявшие собственные загрузочные экраны, нуждались в визуальных эффектах в очень малом количестве байт BASIC – LPRINT был идеален. Техника вышла из употребления, когда сцена перешла на машинный код.

Но в 2011 году JtN и 4D выпустили **BBB** – демо, намеренно вернувшееся к LPRINT как художественному высказыванию. Старый трюк пиратских загрузчиков, оформленный с намерением, стал демо-артом. Ограничение – BASIC, как перенаправления принтера, без машинного кода – стало медиумом.

Почему это важно для sizecoding

LPRINT достигает сложного вывода на экран при почти нулевом расходе твоего собственного кода. ПЗУ делает тяжёлую работу. Твой вклад: POKE для перенаправления вывода, данные для печати и RST \$10 (или LPRINT) для запуска. Ты используешь 16К ПЗУ Spectrum как “бесплатный” движок вывода на экран – код, который не считается в твой лимит размера.

13.5 512-байтные интро: пространство для дыхания

Удвоение с 256 до 512 байт – это не вдвое больше – это качественно другое. На 256 ты борешься за каждую инструкцию, и звук минимален. На 512 ты можешь иметь полноценный эффект и полноценный звук, или два эффекта с переходом.

Что позволяет каждый размерный уровень

Переход между размерными категориями не линейный. Каждое удвоение открывает качественно новые возможности:

256 байт – это один эффект и, может быть, примитивный звук. Ты не можешь позволить себе таблицу данных длиннее примерно 16 байт. Каждая переменная живёт в регистре или в потоке инструкций (самомодифицирующийся код). Текстовый вывод ограничен несколькими символами. У тебя есть место для одного вложенного цикла с 2-3 арифметическими операциями во внутреннем теле. Визуальная часть будет атрибутной, сгенерированной исключительно из арифметики. Звук, если он есть – это свип тона или случайные ноты.

512 байт позволяют добавить таблицу синусов (32-64 байта), настоящий музыкальный движок AY (мелодия + бас на двух каналах) или второй визуальный эффект с переходом. Ты можешь позволить себе полноценный конечный автомат с подсчётом кадров, переключающий между двумя частями. Самомодифицирующийся код (SMC) становится структурным, а не отчаянным. У тебя может даже оставаться место для короткой текстовой строки (10-20 символов), выводимой через RST \$10.

1К (1024 байта) – это другой мир. Ты можешь иметь трекерный проигрыватель музыки со сжатым паттерном (один канал с 32-шаговым циклом занимает около 80-120 байт, включая проигрыватель). Множественные эффекты с переходами становятся нормой. Пиксельные эффекты – простая плазма в пиксельном пространстве, скроллинг текста, растровые полосы – становятся осуществимы, потому что ты можешь позволить себе вычисление адреса экранной памяти. Ты можешь включить 256-байтную таблицу синусов или сгенерировать её при старте и хранить в буфере. На 1К ограничение всё ещё формирует каждое решение, но решения уже о том, *какие возможности включить*, а не о том, *какие инструкции ты можешь себе позволить*.

Интро на **4К и 8К** приближаются к территории коротких демо. На 4К сжатие становится жизнеспособным, и ты можешь вместить многоэффектные композиции с музыкой – качественный скачок, рассмотренный в разделе 13.6. 8К-интро – это отполированное мини-демо, где ограничение больше касается сжатия данных, чем трюков на уровне инструкций. Техники из этой главы по-прежнему применимы, но фокус смещается с “могу ли я сэкономить один байт?” на “могу ли я сжать этот поток данных?”

Оптимальная точка для обучения sizecoding – 256 байт. На этом размере каждая техника из данной главы обязательна. На 512 у тебя достаточно места для выбора. На 1К мышление сайзкодера помогает, но не доминирует.

Типичные 512-байтные паттерны

Плазма через суммы синусов. Таблица синусов – дорогая часть. Полная 256-байтная таблица потребляет половину бюджета. Решения: 64-элементная четвертьволновая таблица, зеркалируемая в рантайме (экономит 192 байта), или генерация таблицы при запуске с использованием параболической аппроксимации из Главы 4 (~20 байт кода вместо 256 байт данных).

Туннель через lookup угла/расстояния. На 512 байт ты вычисляешь угол и расстояние на лету, используя грубые аппроксимации. Визуальное качество ниже, чем у туннеля Eager (Глава 9), но распознаемо как туннель.

Огонь через клеточный автомат. Каждая ячейка усредняет соседей снизу минус затухание. Несколько инструкций на пиксель, убедительная анимация, и на 512 байт можно добавить атрибуты для цвета и звук бипера.

Самомодифицирующиеся трюки

Самомодификация становится структурной на 512 байтах. Встрой счётчик кадров *внутрь* инструкции:

```
frame_ld:
    ld    a, 0           ; this 0 is the frame counter
```

```
inc a
ld (frame_ld + 1), a ; update the counter in place
```

Никакой отдельной переменной. Счётчик живёт в потоке инструкций.

Патч смещений переходов для переключения между эффектами:

```
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
```

Трюк с ORG

Выбирай адрес ORG своей программы так, чтобы байты адреса сами были полезными данными. Размести код по адресу \$4000, и каждый JR/DJNZ, направленный на метки вблизи начала, генерирует маленькие байты смещения – пригодные как счётчики циклов, значения цвета или номера регистров AY. Если твоему эффекту нужен \$40 (старший байт экранной памяти) как константа, размести код по адресу, где \$40 естественно появляется в операнде адреса. *Кодировка самого кода* предоставляет данные, нужные тебе в другом месте.

Это самый глубокий уровень головоломки sizecoding.

13.6 4К-интро: мини-демо

4096 байт – это точка, где sizecoding переходит от “одного трюка” к “мини-демо”. На 256 байтах у тебя есть место для единственного эффекта и, может быть, примитивного звука. На 512 или 1К ты можешь иметь полноценный эффект с музыкой. На 4К ты можешь иметь несколько эффектов, переходы между ними, полный саундтрек и связную повествовательную арку. Разница между 1К и 4К – качественная, а не просто количественная: это разница между “умным трюком” и “крохотной продукцией”.

Сжатие становится жизнеспособным

Самое крупное изменение на 4К – сжатие данных начинает окупаться. Хороший Z80-распаковщик – ZX0, Exomizer или аналогичный – стоит примерно 150-200 байт кода. На 256 или 512 байтах эти накладные расходы катастрофичны. На 4К это менее 5% твоего бюджета, а отдача огромна: 4К-интро может содержать 6-8К несжатых кода и данных, упакованных до допустимого предела. Твоё фактическое рабочее пространство почти удваивается.

Рабочий процесс превращается в цикл обратной связи: пишешь код, ассемблируешь в сырой бинарник, сжимаешь ZX0, проверяешь размер выходного файла, итерируешь. Число, которое теперь имеет значение – это не размер ассемблированного файла, а *сжатый* размер. Это меняет твою стратегию оптимизации. Ты больше не считаешь отдельные байты инструкций. Ты думаешь о том, что хорошо сжимается.

Код с повторяющимися паттернами сжимается лучше, чем код с высокой энтропией. Таблица значений синуса сжимается хорошо (гладкая, предсказуемая). Таблица случайных байт – нет. Код эффекта, повторно использующий похожие последовательности инструкций в разных подпрограммах, сжимается лучше, чем код, где каждая подпрограмма имеет уникальную структуру. Это тонкий сдвиг: ты оптимизируешь не просто ради *маленького кода*, а ради *сжимаемого кода*.

Музыка помещается

На 256 байтах звук – роскошь: свип тона или случайные пентатонические ноты. На 4К ты можешь иметь настоящий саундтрек. Крошечный движок проигрываемеля AY – что-то вроде вывода Beepola или собственного минимального трекера – занимает 200-400 байт. Добавь 500-1000 байт данных паттернов (в сжатом виде), и у тебя полноценная трёхканальная AY-композиция с мелодией, басом и ударными. Эти числа хорошо сжимаются, потому что данные музыкальных паттернов высокоповторямы.

Влияние на аудиторию непропорционально велико. Звук превращает sizecoding-работу из визуального курьёза в *переживание*. На показах компо интро с музыкой набирают значительно больше голосов, чем молчаливые работы равного визуального качества. Если у тебя есть 4К для работы и ты не включаешь музыку, ты упускаешь очки.

Многоэффектная структура

В отличие от 256 байт, где ты привязан к единственному визуалу, 4К даёт место для 2-4 различных эффектов с переходами. Структурный каркас весит немного: таблица сцен, отображающая указатели на эффекты к длительностям, стоит, пожалуй, 30 байт:

```
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

```

Каждый отдельный эффект может занимать 500-1000 байт кода. На 4К в сжатом виде ты можешь позволить себе три существенных эффекта, таблицу сцен, проигрыватель музыки и логику переходов (затемнение в чёрный между сценами стоит дёшево – просто обнуляешь область атрибутов).

GOA4K, inal и Megademica

GOA4K от Exploder^{^XTM} – это знаковое 4К-интроверо для ZX Spectrum 128K, демонстрирующее, чего можно достичь, когда сжатие встречается с умным кодированием. Оно упаковывает чанковый ротозумер и другие эффекты в 4096 байт – визуальная часть, которая выглядела бы достойно в полноразмерном демо, сжатая до размера, который поместился бы в один дисковый сектор.

На этом история не заканчивается. **SerzhSoft** взял GOA4K и переделал его в **inal** – версию только для 48K, всего в 2980 байт. Тот же визуальный удар, на более ограниченной машине, в меньшем количестве байт. Так работает sizecoding-сообщество: один кодер устанавливает планку, другой берёт её с более трудной стартовой позиции.

SerzhSoft продолжил и выиграл компо 4К-интроверо на **Revision 2019** с **Megademica** – соревнуясь не в ZX-специфичной категории, а против всех платформ на крупнейшем демосценовском мероприятии мира. ZX Spectrum 4К-интроверо, судимое наравне с PC и Amiga работами, заняло первое место. Вот траектория, которую открывает sizecoding на 4К: от локальной техники сцены к мировому признанию.

Изучение таких работ выявляет закономерность: лучшие 4К-интроверо выбирают эффекты, которые визуально впечатляющи и хорошо сжимаются, а затем выжимают каждый байт через плотный цикл упаковка-тест-итерация.

Компромиссы на 4К

Работа на 4К вводит компромиссы, которых не существует при меньших размерах:

Степень сжатия определяет выбор эффекта. Не все эффекты сжимаются одинаково. Плазма, опирающаяся на гладкую таблицу синусов, сжимается превосходно – данные таблицы предсказуемы, а цикл рендеринга повторно использует похожие шаблоны инструкций. Эффект псевдослучайного дизеринга,

где каждый пиксель вычисляется по другой формуле, производит код с высокой энтропией, который почти не сжимается. На 4К ты выбираешь эффекты отчасти по их визуальным достоинствам и отчасти по тому, насколько хорошо их реализация пакуется.

Время загрузки заметно. Распаковка занимает реальное время – обычно 1-3 секунды на 3,5 МГц Z80 для нескольких килобайт данных. Зритель видит паузу перед началом интро. Большинство 4К-интров маскируют это простым загрузочным эффектом: заполняют бордюр циклированием цветов, рисуют быстрый паттерн в атрибутах или показывают однокадровый титульный экран. Сам распаковщик работает из небольшого несжатого заглушки в начале файла. Как только распаковка завершается, заглушка прыгает к распакованному коду и начинается настоящее шоу.

Ты оптимизируешь для упакованного размера, а не для скорости выполнения. В 256-байтном интре тот же код, что выполняется – это код, который ты измеряешь. На 4К ты пишешь код, который распаковывается в RAM и затем выполняется оттуда. Ограничения ROM исчезают – твой распакованный код лежит в свободной RAM. Но цель оптимизации смещается: тебя волнует, сколько байт занимает упакованный бинарник, а не сырой ассемблированный размер. Эффект, который ассемблируется в 900 байт, но сжимается до 400 – лучше, чем тот, что ассемблируется в 600, но сжимается до 500.

Подсчёт упакованных байтов. Процесс сборки получает шаг сжатия. Ассемблируй в бинарник, сожми ZX0 (или выбранным тобой упаковщиком), проверь размер выходного файла. С sjasmplus:

```
zx0 build/intro4k.bin build/intro4k.zx0
ls -l build/intro4k.zx0      # this is the number that must be <= 4096
```

Заглушка распаковщика, добавляемая перед финальным файлом, тоже должна помещаться в лимит 4096 байт. Общий файл = заглушка распаковщика + сжатые данные. Типичный распаковщик ZX0 занимает около 70 байт в своей минимальной форме, оставляя примерно 4026 байт для сжатых данных.

Категории соревнований

Демосценовские пати предлагают различные категории с ограничением размера помимо классических 256. Распространённые соревновательные уровни включают 4К, 8К и иногда 16К, наряду с меньшими 256 и 512. Конкретные категории варьируются от пати к пати – Chaos Constructions, DiHalt и Forever все проводили 4К-компо для Spectrum. Некоторые пати объединяют платформы (компо “4К-интровер”, принимающее работы для любой 8-битной платформы), другие специфичны для Spectrum. Проверяй правила пати перед началом работы – метод измерения (сырой размер файла против образа загруженной памяти) и точный лимит в байтах имеют значение.

На 8К и 16К подход по сути тот же, что и на 4К, но с большим простором. 8К-интровер – это отполированное мини-демо, где конвейер сжатия стандартен, а творческий вызов – больше в арт-дирекции, чем в подсчёте байтов. На 16К ты по сути делаешь короткое демо, которое умещается в 16К – ограничение размера формирует твою амбицию, но не диктует выбор инструкций. Техники sizecoding из этой главы по-прежнему помогают при этих больших бюджетах, но их влияние пропорционально меньше.

13.7 Практика: пишем 256-байтное интро пошагово

Начни с работающей атрибутной плазмы (~400 байт) и оптимизириуй до 256.

Шаг 1: Неоптимизированная версия

Простая атрибутная плазма: заполни 768 байт атрибутной памяти значениями из сумм синусов, смещёнными счётчиком кадров. Звук: циклическая мелодия на канале А AY. Эта версия чистая, читаемая и примерно 400 байт – таблица синусов (32 байта), таблица нот (16 байт), онлайн-записи AY и цикл плазмы с табличными поисками.

Шаг 2: Замени CALL на RST

Любой вызов к адресу ПЗУ, совпадающему с вектором RST, экономит 2 байта за каждое использование. Для вывода AY замени шесть многословных онлайн-записей регистров (~60 байт) маленькой подпрограммой:

```
ay_write:           ; register in A, value in E
    ld bc, $FFF
    out (c), a
    ld b, $BF
    out (c), e
    ret          ; 8 bytes total
```

Шесть вызовов (5 байт каждый: загрузка А + загрузка Е + CALL) = 30 + 8 = 38 байт. Экономия: ~22 байта.

Шаг 3: Перекрой данные с кодом

32-байтная таблица синусов в точке входа декодируется как в основном безвредные инструкции Z80 (\$00=NOP, \$06=LD B,n, \$0C=INC C...). Размести её в начале программы. При первом выполнении ЦП спотыкается через эти “инструкции”, портая некоторые регистры. Основной цикл затем перепрыгивает через таблицу и больше не выполняет её – но данные остаются для поиска. Байты таблицы служат двойной цели.

Шаг 4: Используй состояние регистров

После того как цикл плазмы записал 768 атрибутов, HL = \$5B00 и BC = 0 (от любого LDIR, использованного при инициализации). Если следующей операции нужны эти значения, пропусти явные загрузки. Открытие Art-Top в NHBF было именно этим: значения регистров от очистки экрана совпали с длиной текстовой строки. Не запланировано. Замечено.

После каждого прохода оптимизации аннотириуй, что содержит каждый регистр в каждой точке. Состояние регистров – это разделяемый ресурс – фундаментальная валюта sizecoding.

Шаг 5: Меньшие кодировки везде

- LD A, 0 -> XOR A (экономия 1 байт)
- LD HL, nn + LD A, (HL) -> LD A, (nn) (экономия 1 байт, если HL не нужен)
- JP -> JR везде (экономия 1 байт каждый)
- CALL sub : ... : RET -> проваливание напрямую (экономия 4 байта)
- PUSH AF для временного сохранения вместо LD (var), A (экономия 2 байта)

Шаг 6: Точный подсчёт байтов

Интуиция о “сколько это весит” ненадёжна. Нужно считать. Есть три метода, и серьёзные сайзкодеры используют все три.

Вывод ассемблера. sjasmplus может сообщить размер ассемблированного файла. Директива DISPLAY печатает в консоль во время ассемблирования, а ASSERT обеспечивает соблюдение лимита:

```
intro_end:
    ASSERT intro_end - init <= 256, "Intro exceeds 256 bytes!"
    DISPLAY "Intro size: ", /D, intro_end - init, " bytes"
```

Запускай ассемблер после каждого изменения. Стока DISPLAY показывает, где ты находишься; ASSERT ловит превышения до того, как ты потратишь время на тестирование сломанного бинарника.

Анализ файла символов. Ассемблируй с --sym=build/intro.sym, чтобы получить таблицу символов. Сравнивай адреса меток, чтобы определить, сколько байт занимает каждая секция. Когда твоё интро – 262 байта и тебе нужно срезать 6, файл символов покажет, что инициализация AY – 22 байта (можешь ли ты убрать 2?), цикл эффекта – 38 байт (можно ли объединить счётчики строк и столбцов?), обратная запись счётчика кадров – 8 байт (можно ли реструктурировать до 5?). Без этой раскладки ты гадаешь.

Инспекция hex-дампа. После ассемблирования изучи сырой бинарник в hex-редакторе (или xxd build/intro.bin). Hex-дамп показывает тебе реальные байты, которые процессор будет выполнять. Ты заметишь избыточности, невидимые в исходнике: две последовательные загрузки, которые можно объединить в одну, опкод, значение которого совпадает с данными, нужными в другом месте, последовательность NOP, оставленную случайным выравниванием. Hex-дамп – это истина. Исходник – абстракция над ней.

Финальный рывок

Последние 10-20 байт – самые трудные. Структурная перестановка: переставь код так, чтобы проваливания устранили инструкции JR. Объедини звуковой и визуальный циклы. Встрой байты данных в поток инструкций – если тебе нужен \$07 как данные и тебе также нужен RLCA (опкод \$07), устрой так, чтобы один служил обоим.

На этом этапе веди лог. Записывай каждое изменение, которое пробуешь: “переместил инициализацию AY перед заливкой пикселей: сэкономил 2 байта (повторное использование регистра C), потерял 1 байт (нужен дополнительный LD B). Итого: +1 байт.” Многие изменения не помогают. Некоторые делают хуже. Без лога ты попробуешь один и тот же тупик дважды. С логом ты строишь карту пространства решений.

Пробуй радикальную реструктуризацию. Может ли цикл визуального эффекта заодно обновлять AY? Если внутренний цикл итерирует 768 раз (по одному на каждую ячейку атрибутов), и ты записываешь новое значение тона каждые 32 итерации (раз в строку), обновление звука происходит внутри визуального цикла ценой одной проверки BIT 4, E / JR NZ – 4 байта, чтобы объединить две подпрограммы, которым раньше нужен был отдельный каркасный код. Иногда слияние экономит 10 байт; иногда стоит 5. Ты не узнаешь, пока не попробуешь.

Запасной выход: выбери другой эффект. Если твоя плазма требует таблицу синусов и ты на 30 байт превышаешь лимит, никакая микрооптимизация тебя не спасёт. Переключись на эффект, генерирующий визуал из чистой регистровой арифметики: XOR-паттерны, модулярная арифметика, битовые манипуляции. Интерференционный паттерн на основе XOR, как в `intro256.a80`, не требует ни одного байта данных. Визуал менее плавный, чем синусная плазма, но он помещается. На 256 байтах “помещается” – единственный критерий, который имеет значение.

Ты вглядываешься в hex-дамп. Ты пробуешь переместить звуковую подпрограмму перед визуальной. Ты пробуешь заменить таблицу синусов на рантайм-генератор. Каждая попытка перетасовывает байты. Иногда всё сходится.

Удовлетворение от вмещения связного аудиовизуального опыта в 256 байт – от решения головоломки – реальное и особенное, и не похоже ни на какое другое чувство в программировании.

13.8 Sizecoding-музыка: Bytebeat на AY

В PC-демосцене **bytebeat** – это формульный подход к звуку: единственное выражение вроде `t*((t>>12|t>>8)&63&t>>4)` генерирует PCM-сэмплы, производя удивительно сложную музыку из нескольких байт кода. Концепция была популяризирована Viznut (Ville-Matias Heikkilä) в 2011 году, и 256-байтные PC-интро регулярно используют bytebeat для своих саундтреков.

На ZX Spectrum ситуация другая. AY-3-8910 – это не ЦАП, а генератор тона и шума с регистрами периода и громкости для каждого канала. Ты не можешь подавать ему PCM-сэмплы в традиционном смысле (воспроизведение сэмплов через регистр громкости существует, но стоит слишком много тиков для sizecoding-интрос). Вместо этого “AY bytebeat” означает вычисление **периодов тона и огибающих громкости по математическим формулам**, управляемым счётчиком кадров.

Принцип тот же, что и в PC bytebeat: замени хранимые музыкальные данные формулой. Цель вывода другая.

Минимальный формульный движок AY

Типичный подход в 256-байтном интре:

```
; 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

```

Это производит циклический тон, плавающий по периодам и затухающий/нарастающий – не музыку в каком-либо традиционном смысле, но узнаваемо структурированный звук. Трюк в том, чтобы выбрать формулы, производящие **музыкально интересные паттерны** из простых побитовых операций.

Техники для более качественного звука из формул

Пентатоническое маскирование. Сырые побитовые формулы производят хроматический шум. Пропусти значение периода через пентатоническую таблицу подстановки (5 байт: интервалы нот), чтобы ограничить вывод приятной гаммой. Пять байт данных покупают музыкально связный звук.

Многоканальные формулы. AY имеет три тональных канала. Используй разные битовые ротации одного и того же счётчика кадров для каждого канала – они произведут связанные, но различные паттерны, создавая впечатление гармонии:

```

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

```

Шумовая перкуссия. Включай генератор шума на определённых интервалах кадров (каждый 8-й или 16-й кадр) для ритмического пульса. Стоимость: один AND + один OUT – около 6 байт для базового паттерна бочки.

LD A,R как энтропия. Регистр R (счётчик обновления памяти) фактически случаен с музыкальной точки зрения. Смешай его со счётчиком кадров: ld a,r : xor (frame) производит эволюционирующие текстуры, которые никогда полностью не повторяются. Полезно для эмбиентных или экспериментальных звуковых ландшафтов.

Bytebeat против секвенсированной музыки

	Bytebeat (формула)	Секвенсированная (данные паттернов)
Байты	10-30 (только код)	200-400 (проигрыватель) + 500+ (паттерны)

	Bytebeat (формула)	Секвенсированная (данные паттернов)
Музыкальное качество	Абстрактное, генеративное, инопланетное	Мелодичное, структурированное, человеческое
Лучше всего для Звук	256b, 512b Ритмический шум, свины, дроны	1К, 4К Настоящие мелодии

На 256 байтах bytebeat – твой единственный реалистичный вариант: для проигрывателя паттернов нет места. На 512 ты можешь позволить себе крошечный секвенсор с 4-8 нотами. На 4К используй настоящий проигрыватель. Подход bytebeat не уступает – он производит *другой вид* звука, соответствующий эстетике крошечных программ. Некоторые из самых запоминающихся 256-байтных интро запомнились именно потому, что их звук был инопланетным и генеративным, а не потому, что он имитировал конвенциональную музыку.

13.9 Процедурная графика: компо Rendered GFX

Некоторые демосценовские пати проводят соревнование **rendered graphics** (или **procedural graphics**): подай программу, которая генерирует статичное изображение. Никаких предварительно нарисованных битмапов, никаких загруженных данных – каждый пиксель должен быть вычислен. Визуальный результат оценивается как произведение искусства, но он должен быть рождён из кода.

На Spectrum это значит, что твоя программа заполняет 6912-байтную область экрана (битмап + атрибуты) алгоритмически, а затем останавливается. Изображение остаётся на экране для оценки. Лимиты размера файла варьируются – некоторые компо допускают любой размер, другие устанавливают ограничение в 256 байт или 4К, превращая это в гибрид sizecoding и цифрового искусства.

Почему Spectrum интересен для этого

Ограничения дисплея Spectrum – 1-битные пиксели с 8x8 атрибутным цветом – делают процедурную графику подлинно другим вызовом по сравнению с 256-цветным VGA или 24-битным фреймбуфером. Ты не можешь просто вычислить RGB-значения для каждого пикселя. Ты должен думать в терминах:

- **Пиксельные паттерны** внутри 8x8 знакомест (дизеринг, полутон)
- **Атрибутный цвет** на знакоместо (2 цвета из палитры в 15)
- **Взаимодействие** пиксельного паттерна и атрибута – для градиента нужны и плавный дизеринг, И плавные переходы атрибутов

Это ограничение создаёт характерный визуальный стиль. Процедурная графика на Spectrum не похожа ни на что другое – цветовая сетка придаёт ей мозаичное качество, которое является частью эстетики, а не недостатком, который нужно скрывать.

Распространённые подходы

Множества Мандельброта и Жюлиа. Классический выбор. Цикл итерации компактен (~30-50 байт для ядра), а фрактальная детализация бесконечна – координаты масштабирования и количество итераций – единственные параметры. Сопоставь количество итераций с паттерном дизеринга для пиксельных данных, сопоставь с индексом палитры для атрибутов. Рендерер Мандельброта комфортно помещается в 256 байт и производит изображения, которые выглядят нарисованными вручную.

Интерференционные паттерны. Множество наложенных друг на друга синусоидальных или косинусоидальных волн, дискретизированных в каждой позиции пикселя. $\text{pixel} = \sin(x*\text{freq1} + \text{phase1}) + \sin(y*\text{freq2} + \text{phase2}) > \text{threshold}$. Производит органические, текущие формы. На Spectrum пороговое значение суммы даёт пиксельный бит, квантование даёт атрибутный цвет.

Поля расстояний. Вычисли расстояние от каждого пикселя до набора фигур (кругов, линий, кривых Безье). Пороговое значение расстояния для пиксельных данных, отображение на цвет для атрибутов. Несколько фигур могут произвести удивительно сложные изображения – одних наложенных кругов достаточно для создания замысловатых паттернов.

L-системы и фракталы. Рекурсивные ветвящиеся структуры (деревья, папоротники, треугольники Серпинского). Рекурсия естественно ложится на стековый Z80-код, а визуальный результат имеет органическую сложность при минимальном коде. Рендерер треугольника Серпинского – это около 20 байт; ветвящееся дерево со случайными углами – пожалуй, 80.

Байтовый бюджет для искусства

В компо rendered GFX с ограничением размера каждый байт идёт на визуальную сложность. Нет цикла кадров, нет звука, нет анимации – только прямолинейная программа, заполняющая экран и останавливающаяся. Это значит, что весь твой бюджет идёт на код рендеринга и генерацию координат. На 256 байтах ты можешь произвести детальный фрактал. На 4K (со сжатием) ты можешь генерировать изображения с множеством слоёв, вычисленными текстурами и тщательным дизерингом, приближающимся к качеству ручной работы.

Критерий оценки – чисто визуальный: зрители голосуют за изображение, а не за код. Но ограничение по коду формирует эстетику. Процедурная графика на Spectrum имеет узнаваемый вид: математическая точность, фрактальная детализация и характерная цветовая сетка атрибутного рендеринга. Лучшие работы принимают эти ограничения как стиль, а не борются с ними.

13.10 Sizecoding как искусство

Sizecoding учит вещам, которые улучшают всё твоё кодирование: дисциплина сомнения в каждом байте обостряет осознание кодирования инструкций, привычка искать перекрытия переносится на любую оптимизационную работу, а

практика использования начального состояния и побочных эффектов делает тебя лучшим системным программистом.

Итого

- **Sizecoding**-соревнования требуют полных программ в 256, 512, 1К, 4К или 8К байт – строгие лимиты, требующие принципиально другого подхода к программированию.
- **Инструментарий сайзкодера** включает предположения об инициализации регистров, DJNZ как совмещённый декремент-и-переход, RST как 1-байтный CALL, перекрывающиеся инструкции и эксплуатацию флагов через SBC A,A – трюки, экономящие 1-5 байт каждый, но накапливающиеся по всей программе.
- **NHBF** (UriS, CC 2025) демонстрирует мышление на 256 байт: каждый байт работает за двоих, состояния регистров одной подпрограммы питают следующую, выбор инструкций диктуется исключительно размером кодировки.
- **Байтовый бюджет** типичного 256-байтного интро выделяет ~90-130 байт на каркас (заливка экрана, инициализация AY, синхронизация с кадром, структура цикла), оставляя 120-160 байт на собственно творческий эффект.
- **Выбор правильного эффекта** важнее микрооптимизации: атрибутные визуалы с арифметическими формулами (XOR, модульная математика) кодируются дёшево; пиксельные эффекты и таблицы данных потребляют слишком много байт на 256.
- **Трюк LPRINT** (dver4d, 2015) перенаправляет вывод принтера BASIC на экранную память через адрес 23681, производя сложные визуальные паттерны в считанных байтах – от пиратских загрузчиков с кассет до демо-арта.
- **Каждый размерный уровень качественно отличается:** 256 байт позволяют один эффект с минимальным звуком; 512 добавляют таблицы синусов и двухканальную музыку; 1К открывают пиксельные эффекты, трекерную музыку и несколько частей; 4К пересекают порог территории мини-демо со сжатием, полными саундтреками и многоэффектными композициями.
- **4К-интро** – это порог, где сжатие становится жизнеспособным: ~200-байтный распаковщик разблокирует 6-8К рабочего пространства, проигрыватели музыки с данными паттернов комфортно помещаются, а таблицы сцен позволяют создать 2-4 различных эффекта с переходами. Цель оптимизации смещается от сырого ассемблированного размера к сжатому упакованному размеру.
- **AY bytebeat** заменяет хранимые музыкальные данные формулами: вычисляя периоды тона и громкости из счётчика кадров, используя побитовую арифметику. На 256 байтах формульный звук (10-30 байт) – единственный вариант; на 4К переключайся на настоящий проигрыватель паттернов. Пентатоническое маскирование, многоканальная битовая ротация и шумовая перкуссия добавляют музыкальности за минимум байтов.
- **Процедурная графика** (rendered GFX) – соревнования, требующие вычисления каждого пикселя, а не его загрузки. 1-битные пиксели Spectrum с

8x8 атрибутным цветом делают это уникальным вызовом – множества Мандельброта, интерференционные паттерны, поля расстояний и L-системы – все они дают характерные результаты в эстетике атрибутной сетки.

- **Процесс оптимизации** идёт от структурных изменений (устранение таблиц, слияние циклов) к выбору кодировок (RST вместо CALL, JR вместо JP, XOR A вместо LD A,0) к случайным находкам (состояния регистров, совпадающие с потребностями в данных).
 - **Точный подсчёт байтов** – через DISPLAY/ASSERT ассемблера, анализ файла символов и инспекцию hex-дампа – необходим. Интуиция о размере кода ненадёжна.
 - **Трюк с ORG** – выбор адреса загрузки так, чтобы байты адреса служили полезными данными – представляет самый глубокий уровень головоломки.
-

Попробуй сам

1. **Начни большим, ужми маленьким.** Напиши атрибутную плазму со счётчиком кадров. Заставь работать на любом размере. Затем оптимизируй до 512 байт, отслеживая каждый сэкономленный байт и как.
 2. **Исследуй LPRINT.** В BASIC попробуй POKE 23681,64 : FOR i=1 TO 500 : LPRINT CHR\$(RND*96+32); : NEXT i. Наблюдай, как транспонированные данные заполняют экран. Экспериментируй с разными диапазонами символов.
 3. **Картографируй состояние регистров.** Напиши маленькую программу и аннотируй, что содержит каждый регистр в каждой точке. Ищи места, где выход одной подпрограммы совпадает с нужным входом другой.
 4. **Изучи векторы RST.** Дизассемблируй ПЗУ Spectrum по адресам \$0000, \$0008, \$0010, \$0018, \$0020, \$0028, \$0030, \$0038. Это твои “бесплатные” подпрограммы.
 5. **Вызов на 256 байт.** Доведи практику из этой главы до 256 байт. Тебе придётся принимать трудные решения о том, что оставить, а что отбросить. В этом и суть.
-

Далее: Глава 14 – Сжатие: больше данных в меньшем пространстве. Мы переходим от программ, умещающихся в 256 байт, к задаче вмещения килобайтов данных в килобайты хранилища, с исчерпывающим бенчмарком Introspec'а 10 упаковщиков в качестве руководства.

Источники: UriS “NHBF Making-of” (Hype, 2025); diver4d “LPRINT Secrets” (Hype, 2015)

Глава 14: Сжатие — больше данных в меньшем пространстве

ZX Spectrum 128K имеет 128 килобайт ОЗУ. Это звучит щедро, пока ты не начнёшь вычитать: экран забирает 6 912 байт (6 144 пикселей + 768 атрибутов), системные переменные претендуют на свою долю, музыкальному проигрывателю AY и его данным паттернов нужен банк-другой, твой код занимает ещё несколько тысяч байт, и стеку нужно пространство для дыхания. К тому моменту, когда ты садишься хранить фактическое содержимое своего демо — графику, кадры анимации, предрассчитанные таблицы подстановки — ты борешься за каждый байт.

Одно полноэкранное изображение на Spectrum — это 6 912 байт. 4K интро может вместить примерно 0,6 одного. 48K демо теоретически могло бы содержать семь экранов без ничего другого. Но демо — не слайд-шоу. В них есть музыка. Есть код. Есть эффекты, требующие таблиц предрассчитанных данных. Вопрос не в том, сжимать ли — а какой упаковщик использовать и когда.

Эта глава построена вокруг бенчмарка. В 2017 году Introspec (spke, Life on Mars) опубликовал “Data Compression for Modern Z80 Coding” на Hype — скрупулёзное сравнение десяти инструментов сжатия, протестированных на тщательно разработанном корпусе. Та статья с её 22 000 просмотров и сотнями комментариев стала справочником, к которому обращаются ZX-кодеры при выборе упаковщика. Мы пройдём через его результаты, поймём компромиссы и научимся выбирать правильный инструмент для каждой задачи.

Проблема памяти

Давай будем конкретны насчёт ограничений. Рассмотрим Break Space от Thesuper (Chaos Constructions 2016, 2-е место) — демо с 19 сценами, работающее на ZX Spectrum 128K. Одна из этих сцен, Magen Fractal от psndcj, показывает 122 кадра анимации. Каждый кадр — полный 6 912-байтный экран. Без сжатия это 843 264 байта — более чем в шесть раз больше общего ОЗУ машины.

psndcj сжал все 122 кадра в 10 512 байт. Это 1,25% от оригинального размера. Вся анимация, каждый её кадр, помещается в пространство меньше двух несжатых экранов.

Другая сцена в Break Space, анимация Mondrian, упаковывает 256 нарисованных вручную кадров — каждый квадрат вырезан отдельно, индивидуально сжат — в 3 килобайта.

Это не теоретические упражнения. Это продакшн-техники из демо, участвовавшего в одном из самых престижных комп сцены. Сжатие — не оптимизация, применяемая в конце. Это фундаментальное архитектурное решение, определяющее, что может содержать твоё демо.

Сжатие как усилитель пропускной способности

Introspec сформулировал идею, возвышающую сжатие от трюка хранения до техники производительности: **сжатие действует как метод увеличения эффективной пропускной способности памяти**.

Допустим, эффекту нужно 2 КБ данных на кадр. Со храни их сжатыми до 800 байт и распакуй с помощью LZ4 при 34 тактах (T-state) на выходной байт. Распаковка стоит 69 632 такта — почти ровно один кадр. Но ты можешь перекрыть её со временем бордюра, буферизовать кадр вперёд с двойной буферизацией и чередовать срендерингом эффекта. Результат: больше данных проходит через систему, чем шина могла бы доставить из несжатого хранилища. Распаковщик — усилитель данных.

Бенчмарк

Introspec не просто прогнал каждый упаковщик на нескольких файлах и оценил результаты на глаз. Он разработал корпус и измерил систематически.

Корпус

Тестовые данные составили 1 233 995 байт в пяти категориях:

- **Calgary corpus** — стандартный академический бенчмарк сжатия (текст, бинарные, смешанные)
- **Canterbury corpus** — более современный академический стандарт
- **30 графических файлов ZX Spectrum** — загрузочные экраны, мультиколорные изображения, игровые экраны
- **24 музыкальных файла** — РТЗ-паттерны, дампы регистров AY, данные сэмплов
- **Разнообразные ZX-данные** — тайловые карты, таблицы подстановки, смешанные данные демо

Этот микс имеет значение. Упаковщик, превосходный на английском тексте, может буксовать на ZX-графике, где длинные ряды нулей в пиксельной области чередуются с почти случайными атрибутными данными. Тестирование на реальных данных Spectrum — данных, которые ты реально будешь сжимать — необходимо.

Результаты

Десять инструментов. Измерены по общему сжатому размеру (меньше — лучше), скорости распаковки в тактах (T-state) на выходной байт (меньше — быстрее) и размеру кода распаковщика в байтах (меньше — лучше для sizecoding-продукций).

Инструмент	Сжатый (байт)	Степень сжатия	Скорость (T/байт)	Размер распаковщика	Примечания
Exomizer	596 161	48,3%	~250	~170 байт	Лучшая степень сжатия
ApLib	606 833	49,2%	~105	199 байт	Хороший баланс
PuCrunch	616 855	50,0%	—	—	Сложная альтернатива LZ
Hrust 1	613 602	49,7%	—	—	Перемещаемый стеко-вый распаковщик
Pletter 5	635 797	51,5%	~69	~120 байт	Быстрый + приличное сжатие
MegaLZ	636 910	51,6%	~130	~110 байт	Воскрепшён Introspec'ом в 2019
ZX7	653 879	53,0%	~107	69 байт	Крошечный распаковщик
ZX0	—	~52%	~100	~70 байт	Преемник ZX7
LZ4	722 522	58,6%	~34	~100 байт	Самая быстрая распаковка
Hrum	—	~52%	—	—	Объявлен устаревшим

Только Exomizer пробил барьер 600 000 байт по всему корпусу. Но скорость

распаковки Exomizer — примерно 250 тактов (T-state) на выходной байт — делает его непрактичным для всего, что нужно распаковывать во время воспроизведения.

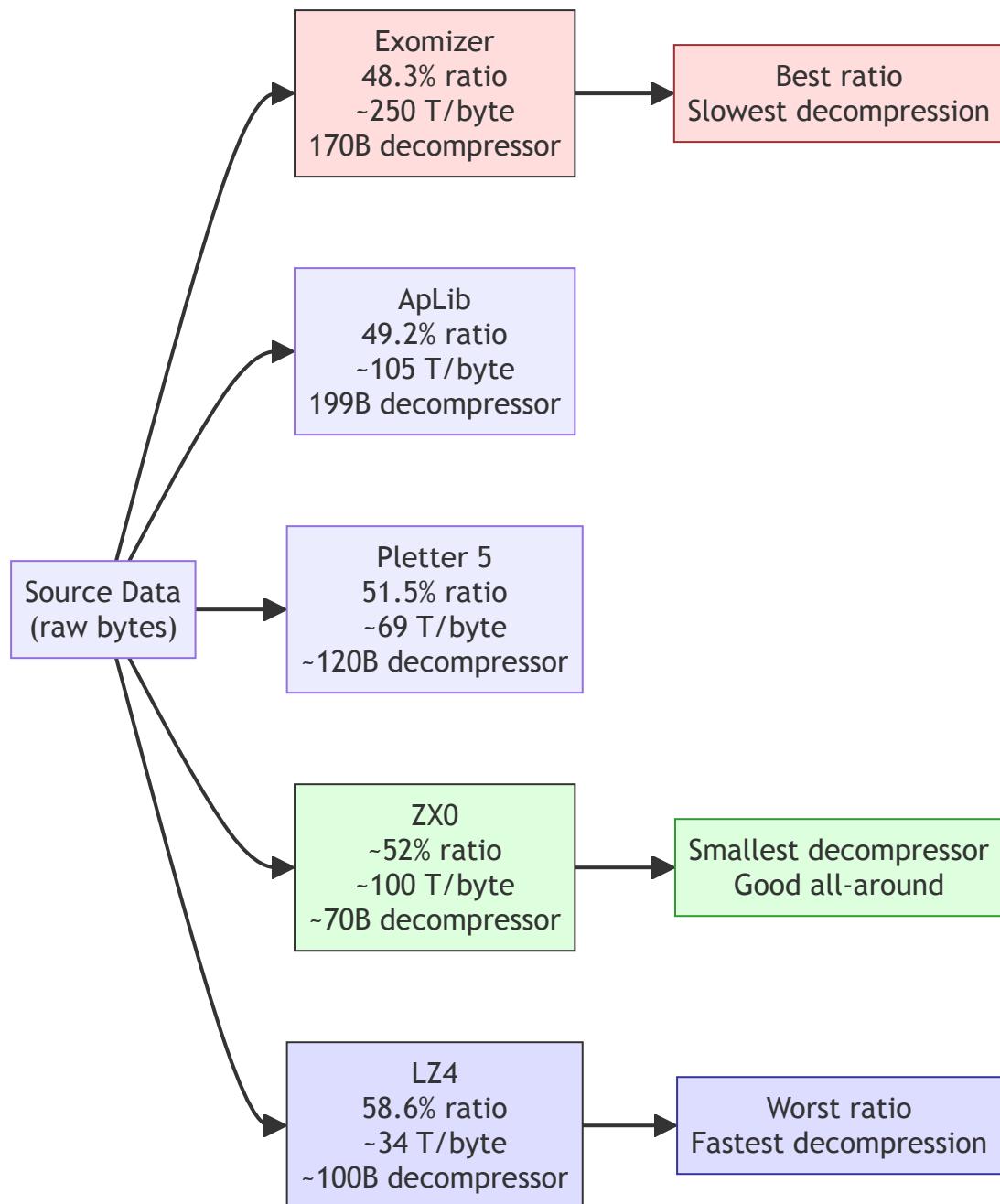
Треугольник компромиссов

Каждый упаковщик делает компромисс между тремя качествами:

1. **Степень сжатия** — насколько маленькими становятся сжатые данные
2. **Скорость распаковки** — сколько тактов (T-state) на выходной байт
3. **Размер кода распаковщика** — сколько байт занимает подпрограмма распаковки

Ты не можешь получить все три. Exomizer побеждает по степени сжатия, но медленно распаковывается и имеет большой распаковщик. LZ4 — самый быстрый для распаковки, но теряет 10 процентных пунктов степени сжатия. ZX7 имеет 69-байтный распаковщик, но сжимает менее агрессивно, чем Exomizer.

Гений Introspec'а в том, что он отобразил эти компромиссы на Парето-фронтре — кривой, где ни один инструмент не может улучшиться по одному измерению без потери по другому. Если упаковщик доминирует по всем трём осям другим инструментом, он устарел. Если он лежит на фронтире — он правильный выбор для какого-то сценария.



Компромисс: Меньший размер сжатых данных = более медленная распаковка. Ни один упаковщик не побеждает по всем трём осям (степень сжатия, скорость, размер распаковщика). Выбирай исходя из задачи: Exomizer для одноразовой загрузки, LZ4 для потокового чтения в реальном времени, ZX0 для sizecoding-интро.

Его практические рекомендации чётки:

- **Максимальное сжатие, скорость не важна:** Exomizer. Используй для однократной распаковки при загрузке — загрузочные экраны, данные уровней, всё, что ты распаковываешь один раз в буфер и используешь многократно.
- **Хорошее сжатие, умеренная скорость (~105 Т/байт):** ApLib. Надёж-

ный универсальный выбор, когда нужна приличная степень и можно позволить ~105 тактов на байт.

- **Быстрая распаковка (~69 Т/байт):** Pletter 5. Когда нужно распаковывать во время геймплея или между сценами и нельзя позволить медленную распаковку Exomizer.
- **Самая быстрая распаковка (~34 Т/байт):** LZ4. Единственный выбор для потоковой передачи в реальном времени — распаковки данных по мере их воспроизведения. При 34 тактах (T-state) на выходной байт LZ4 может распаковать более 2 000 байт за кадр. Это труба данных 2 КБ/кадр.
- **Самый маленький распаковщик (69-70 байт):** ZX7 или ZX0. Когда сам распаковщик должен быть крошечным — в 256-байтных, 512-байтных или 1К интро, где каждый байт кода на счету.

Пусть эти числа направляют твои решения. Нет универсально “лучшего” упаковщика. Есть только лучший упаковщик для твоих конкретных ограничений.

Как работает LZ-сжатие

Все упаковщики в таблице выше принадлежат к семейству Лемпеля-Зива. Понимание основной идеи поможет предсказать, какие данные хорошо сжимаются, а какие нет.

LZ-сжатие заменяет повторяющиеся последовательности байт обратными ссылками. Совпадение говорит: “скопируй N байт с позиции P байт назад в уже декодированном потоке”. Сжатый поток чередуется между **литералами** (сырые байты без полезного совпадения) и **совпадениями** (пары смещение + длина, ссылающиеся на более ранний вывод).

Различия между упаковщиками сводятся к кодированию: сколько бит на смещение, сколько на длину, как сигнализировать литерал против совпадения. Exomizer использует сложные побитовые коды переменной длины, которые сжимают плотно, но требуют тщательного извлечения бит для декодирования — отсюда ~250 тактов (T-state) на байт. LZ4 использует байт-выровненные токены, которые Z80 обрабатывает простыми сдвигами и масками — отсюда ~34 такта на байт ценой 10 процентных пунктов степени сжатия. ZX0 использует однобитные флаги (0 = литерал, 1 = совпадение) с чередующимися кодами Элиаса для длин, попадая в точку баланса между размером и скоростью.

Данные ZX Spectrum хорошо сжимаются, потому что имеют структуру: большие области идентичных байт (чёрные фоны, пустые атрибуты), повторяющиеся паттерны (тайлы, шрифты, интерфейс), коррелированные пиксельные данные с регулярными смещениями. Музыка тоже хорошо сжимается — РТЗ-паттерны полны повторяющихся нотных последовательностей и пустых строк. Что плохо сжимается: случайные данные, уже сжатые данные и очень короткие файлы, где накладные расходы кодирования превышают экономию.

ZX0 — выбор sizecoding-мастера

ZX0, созданный Einar Saukas, – духовный преемник ZX7 и стал стандартным упаковщиком для современной ZX Spectrum-разработки. Он заслуживает особых внимания.

Почему ZX0 существует

ZX7 уже был замечателен: 69-байтный распаковщик, достигающий уважаемой степени сжатия. Но Saukas увидел возможность для улучшения. ZX0 использует алгоритм оптимального парсинга — он не просто находит хорошие совпадения, он находит *наилучшую возможную последовательность* совпадений и литералов для всего файла. Результат — степень сжатия, близкая к значительно более крупным упаковщикам, с распаковщиком, остающимся в диапазоне 70 байт.

Распаковщик

Z80-распаковщик для ZX0 — это вручную оптимизированный ассемблер, спроектированный специально под набор инструкций Z80. Он использует регистр флагов Z80, инструкции блочной пересылки и точные тайминги условных переходов, чтобы выжать максимум функциональности в минимум байт. Вот о каком коде идёт речь:

```
; 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 ...
```

Каждая инструкция работает на два фронта. Аккумулятор служит одновременно битовым буфером и рабочим регистром. Стек хранит последнее использованное смещение для повторных совпадений. Инструкция LDIR обрабатывает и копирование литералов, и копирование совпадений, сохраняя код маленьким.

При примерно 70 байтах весь распаковщик занимает меньше места, чем одна строка символов ZX Spectrum. Для 256-байтного интро это оставляет 186 байт на всё остальное — эффект, анимацию, музыку. Для 4К интро 70 байт — ничтожные накладные расходы. Вот почему ZX0 стал повсеместным.

Когда использовать ZX0

- **256-байтные - 1К интро:** Крошечный распаковщик незаменим. Каждый байт, сэкономленный на распаковщике — байт, доступный для контента.
- **4К интро:** ZX0 может распаковать 4 096 байт в 15-30 КБ кода и данных. Megademica от SerzhSoft (1-е место, Revision 2019) использовала именно эту стратегию, чтобы вместить то, что рецензенты называли “полноценным new-school демо”, в 4К интро.
- **Общая разработка демо и игр:** Когда нужен надёжный универсальный упаковщик с компактным распаковщиком. ZX0 — не самый быстрый распаковщик, но достаточно быстрый для однократной распаковки при загрузке, а его степень сжатия конкурентна с инструментами, имеющими значительно более крупные распаковщики.
- **RED REDUX (2025)** использовал более новый вариант ZX2 (тоже от Saukas) для достижения замечательного подвига — включения Protracker-музыки в 256-байтное интро.

ZX0 — не правильный выбор для потоковой передачи в реальном времени (используй LZ4) или для максимального сжатия любой ценой (используй Exomizer). Но для подавляющего большинства проектов ZX Spectrum он является правильным вариантом по умолчанию.

RLE и дельта-кодирование

Не всему нужен полноценный LZ-упаковщик. Две более простые техники обрабатывают определённые типы данных более эффективно.

RLE: кодирование длин серий

Простейшая схема: замени серию идентичных байт на счётчик и значение. Распаковщик тривиален:

```
; 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
```

Всего 12 байт кода распаковщика. RLE отлично сжимает, когда данные содержат длинные серии — пустые экраны, одноцветные фоны, заливки атрибутов. Он ужасно сжимает сложный пиксель-арт. Преимущество перед LZ: для sizecoding-интров, где даже 70 байт ZX0 кажутся дорогими, 12-байтная RLE-схема высвобождает ценное пространство.

RLE также выигрывает от **транспонирования данных**: если твои данные — двумерный блок (например, 32×24 атрибута), где столбцы более однородны, чем строки, транспонирование в столбцовый порядок создаёт более длинные серии. Стоимость — обратное транспонирование после распаковки (~13 тактов/байт). Даёт ли суммарный результат (12-байтный распаковщик + код обратного транспонирования + сжатые данные) выигрыш перед ZX0 (70-байтный распаковщик + сжатые данные), зависит от твоих данных — измерь оба варианта.

Врезка: Самомодифицирующийся RLE от Ped7g — 9 байт, которые перезаписывают сами себя

Для 256-байтных интров даже 12 байт кажутся дорогими. Ped7g (Peter Helcmanovsky, мейнтейнер sjasmplus) предложил **самомодифицирующийся RLE-распаковщик**, сжимающий декодер до **9 байт основного кода** — а механизм выхода встроен в поток данных.

Трюк: RLE-данные располагаются в памяти *перед* кодом распаковщика. Поток данных завершается байтами \$18, \$00, которые распаковщик записывает в целевой буфер на вычисленную позицию так, что эти байты перезаписывают инструкцию `ld (hl), c`. Последовательность байт \$18, \$23 ассемблируется как `jr +$23`, что выполняет переход вперёд через распаковщик в основной код интров. Данные буквально перезаписывают код, чтобы завершить свою работу.

Вот полноценное мини-интров — 120-байтный бинарник, заполняющий экран цветными полосами исключительно средствами самомодифицирующегося RLE:

```
“‘z80 id:ch14_ped7g_rle_mini_intro ; Ped7g’s self-modifying RLE mini-intro ; Assemble with sjasmplus: sjasmplus rle_intro.a80 ; ; The RLE data is a stream of (value, count) pairs read via POP BC. ; SP walks through the data as a read pointer. ; The db $18,$00 at the end of the data stream overwrites ld (hl),c ; to become jr +$23, exiting the depack loop into intro_start. ; ; Contributed by Ped7g (Peter Helcmanovsky) — sjasmplus maintainer ; and ZX Spectrum Next contributor. Used with permission.
```

```
DEVICE ZXSPECTRUM48, $8000
```

```
target EQU $4000 ORG $5B00 ; loading address → print buffer
```

```
intro_data: dw target ; initial HL value (POP HL) ; RLE pairs: value, count (count=0 means 256 iterations) .(43) db $AA, 0, $00, 0 ; alternating stripe pattern db $43, 322, $44, 324, $45, 323, $46, 322, $47, 322 db $46, 322, $45, 323, $44, 324, $43, 322 db $18, $00 ; data that will overwrite ld (hl),c ; creating jr rle_loop_inner+$25 rle_start: ei ; simulate post-LOAD BASIC environment ld sp, intro_data pop hl ; HL = target address rle_loop_outer: pop bc ; C = value, B = repeat count rle_loop_inner: ld (hl), c ; ← THIS instruction gets overwritten inc hl ; by the $18,$00 data to become djnz rle_loop_inner ; jr +$23, jumping to intro_start jr rle_loop_outer ; 31
```

```

bytes of space — fill with helper code ds $1F intro_start: assert $ ==
rle_loop_inner + 2 + $23 inc a and 7 out (254), a ; cycle border colours jr
intro_start

SAVESNA "rle_intro.sna", rle_start
SAVEBIN "rle_intro.bin", intro_data, $ - intro_data

"""

```

Анализ размера. Цикл распаковки занимает 9 байт: `pop bc (1) + ld hl, c (1) + inc hl (1) + djnz (2) + jr (2) + pop hl (1) + ld sp, nn (3)` = 9 основных + 6 на инициализацию = **15 байт суммарно** для самодостаточного RLE-декодера со встроенным выходом. Сравни с 12-байтным минимальным RLE из предыдущего раздела, которому всё ещё нужна внешняя инициализация и проверка завершения.

Безопасность при прерываниях. SP используется как указатель на данные, поэтому прерывания испортят стек. ei в начале поставлен намеренно — в 256-байтном интро, загруженном из BASIC, прерывания уже разрешены. Случайное прерывание записывает в уже прочитанные данные позади указателя SP, поэтому распаковка завершается корректно. Для кода самого интро SP уже прошёл мимо данных, и стек работает нормально. Но не комбинируй эту технику с IM2 или музыкой на прерываниях.

Продвинутые варианты. Ped7g отмечает несколько альтернативных стратегий выхода: (1) если целевая область простирается за код распаковщика, RLE-данные могут перезаписать смещение `jr rle_loop_outer` для прыжка дальше; (2) трюк с `jp $C3C3` — размести значения \$C3 в данных с точно подобранными счётчиками, чтобы DJNZ завершился, когда в памяти сложится `jp $C3C3`, и выровняй интро так, чтобы адрес \$C3C3 указывал на продолжение кода. Как говорит Ped7g: «таких штук можно изобрести много — всё зависит от конкретной ситуации».

Авторство: Предоставлено Ped7g (Peter Helcmanovsky) — мейнтайнер sjasmplus и контрибьютор ZX Spectrum Next. Используется с разрешения.

Дельта-кодирование: храни то, что изменилось

Дельта-кодирование хранит разности между последовательными значениями, а не абсолютные значения. Два кадра анимации, идентичные на 90%? Храни только изменённые байты — список пар (позиция, новое_значение). Если только 691 байт отличаются из 6 912, дельта составляет 2 073 байта (3 байта на изменение) вместо полного кадра. Примени LZ поверх дельта-потока, и он сжимается ещё сильнее — поток разностей содержит больше нулей и повторяющихся малых значений, чем сырье данные кадра.

Magen Fractal из Break Space использует именно это: 122 кадра по 6 912 байт каждый, сжатые до 10 512 байт суммарно, потому что каждый кадр отличается от предыдущего на малую величину. Дельта + LZ — стандартный конвейер для многокадровых анимаций, скроллящихся тайловых карт и спрайтовых анимаций, где фигура меняет позу, но фон остаётся неподвижным.

Подготовка данных перед сжатием

Дельта-кодирование — не единственный трюк. Упаковщик видит лишь поток байтов, который ты ему подаёшь. Если реструктурировать данные перед сжатием, один и тот же LZ-алгоритм может достичь кардинально разных степеней сжатия. Это искусство предварительной подготовки данных — и зачастую оно ценнее, чем смена упаковщика.

Энтропия: теоретический минимум

Энтропия Шеннона измеряет минимальное количество бит на байт, необходимое для представления твоих данных при идеальном кодировщике. Полностью случайный поток байтов имеет энтропию 8,0 бит/байт — несжимаем. Файл из одинаковых байтов имеет энтропию 0,0. Реальные данные Spectrum находятся где-то между. Сырая таблица синусов может иметь энтропию 6,75 бит/байт. Примени дельта-кодирование, и она упадёт до 2,85. Примени вторую производную, и она снизится до 1,49 — сокращение на 78%. Это теоретический запас, с которым упаковщик может работать.

Тебе не нужно вычислять энтропию вручную. Формула достаточно проста для Python-скрипта:

```
from collections import Counter

def entropy(data: bytes) -> float:
    """Shannon entropy in bits per byte. Lower = more compressible."""
    counts = Counter(data)
    n = len(data)
    return -sum(c/n * math.log2(c/n) for c in counts.values())
```

Запусти это на сырых данных, затем на данных с дельта-кодированием, затем на транспонированных данных. Преобразование, дающее наименьшую энтропию, сожмётся лучше всего, независимо от используемого упаковщика.

Вторая производная: синусоидальные и квадратичные данные

Дельта-кодирование хранит первые разности: $d[i] = data[i] - data[i-1]$. Для линейной рампы (0, 3, 6, 9...) дельта-поток постоянен (3, 3, 3...) — идеально для сжатия. Но синусоиды и гладкие кривые дают дельта-поток, который сам по себе меняется плавно. Вторая производная (дельта от дельты) ловит это:

Тип данных	Сырая энтропия	1-я производная	2-я производная
Таблица синусов (256 байт)	6,75	2,85	1,49
Линейная рампа	7,00	0,00	0,00
Квадратичная кривая	6,80	3,20	0,00
Случайные байты	8,00	8,00	8,00

Вторая производная квадратичной функции — константа. Это не абстрактное исчисление — это разница между 6,80 и 0,00 бит на байт. 256-байтная квадра-

тичная таблица подстановки, закодированная второй производной, сжимается почти в ничто.

Вот ключевая творческая идея: синусоидальное затухание и квадратичное затухание часто визуально неразличимы в демоэффекте. Если ты анимируешь частицу, которая замедляется, зритель не отличит $\sin(t)$ от $at^2 + bt + c$. А вот упаковщик — отличит: квадратичная версия имеет идеально линейную первую производную и постоянную вторую. Если твоя анимация допускает квадратичную аппроксимацию, ты экономишь байты не сменой упаковщика, а сменой кривой.

Транспонирование: столбцовый порядок для табличных данных

Демосценовые данные часто табличные — таблицы 3D-вершин (X, Y, Z на вершину), ключевые кадры анимации (угол, радиус, скорость на кадр), цветовые палитры (R, G, B на запись). При хранении построчно ($X_0 \ Y_0 \ Z_0 \ X_1 \ Y_1 \ Z_1 \dots$) соседние байты принадлежат разным столбцам с разными статистическими свойствами. Дельта-кодирование делает ситуацию хуже:

```
Row-major: 128 64 200 129 63 201 130 62 202 ...
Delta:      64 136 57 190 138 57 190 138 ... (wild jumps between columns)
```

Транспонируй в столбцовый порядок ($X_0 \ X_1 \ X_2 \dots \ Y_0 \ Y_1 \ Y_2 \dots \ Z_0 \ Z_1 \ Z_2 \dots$), и теперь соседние байты принадлежат одному столбцу. Дельта-кодирование видит плавные прогрессии:

```
Column-major: 128 129 130 131 ... 64 63 62 61 ... 200 201 202 203 ...
Delta:        1   1   1   1 ... -1  -1  -1 ...     1   1   1   1 ... (trivial)
```

Цифры впечатляют. Таблица вершин на 768 байт (256 вершин \times 3 столбца):

Раскладка	Энтропия (сырая)	Энтропия (дельта)
Построчная (X, Y, Z чередуются)	7,52	7,66 (хуже!)
Столбцевая, шаг 3	7,52	2,58

Дельта-кодирование построчных данных *увеличило* энтропию. То же самое дельта-кодирование на транспонированных данных уменьшило её на 65%. Упаковщик не знает, что твои данные табличные — ты должен сообщить ему это, переупорядочив данные.

Правило: если в твоих данных есть столбцы с разными паттернами, **всегда транспонируй перед сжатием**. Шаг (количество столбцов) не нужно угадывать — попробуй несколько делителей длины данных и выбери тот, что даёт наименьшую дельта-энтропию.

На Spectrum распаковщик просто записывает байты последовательно. Транспонирование происходит в твоих инструментах сборки, а не во время выполнения. Нулевые затраты во время выполнения.

Чередование плоскостей: маски и пиксели

Спрайты с масками — частный случай транспонирования. При хранении маска-пиксель-маска-пиксель построчно соседние байты чередуются между двумя

совершенно разными распределениями (маски — в основном \$FF или \$00; пиксели имеют разнообразные значения). Раздели все байты масок и все байты пикселей:

Before: FF 3C FF 18 FF 00 ... (mask, pixel, mask, pixel)
After: FF FF FF ... 3C 18 00 ... (all masks, then all pixels)

Блок масок сжимается почти в ничто (длинные серии \$FF). Блок пикселей сжимается как обычно. Совокупная степень сжатия улучшается на 10-20% по сравнению с чередующимся хранением, в зависимости от сложности спрайта.

Обнаружение паттернов: когда не сжимать

Иногда данные имеют структуру, которую генератор может воспроизвести дешевле, чем распаковщик. Если твои данные периодичны с периодом P , хранение одного периода плюс крошечный цикл воспроизведения занимает $P + \sim 10$ байт. Если P мало относительно общего объёма данных, это побьёт любой упаковщик.

Таблицы синусов — канонический пример. 256-байтная таблица синусов сжимается до ~ 140 байт с ZX0. Но спектрумовский генератор синусов (через ROM-калькулятор или CORDIC-ядро) производит те же 256 байт из менее чем 30 байт кода. Для демосценовой точности даже простая квадратичная аппроксимация на четверть волны достаточна.

Дерево принятия решений: (1) Можешь ли ты сгенерировать данные по формуле в меньшее число байт, чем сжатый размер? Генерируй. (2) Данные периодичны? Храни один период + цикл. (3) Данные табличные? Транспонирование + дельта + LZ. (4) Данные — последовательные кадры? Дельта + LZ. (5) Ничего из вышеперечисленного? Просто сжимай.

Практические преобразования для типичных демосценовых данных

Тип данных	Лучшее преобразование	Почему
Таблицы синусов/косинусов	2-я производная, или генерация во время выполнения	Плавное ускорение → постоянная 2-я производная
Таблицы 3D-вершин	Транспонирование (шаг = полей на вершину) + дельта	Разделяет оси; плавные траектории по каждой оси
Предвычисленная анимация	Дельта между кадрами + LZ	Высокая межкадровая избыточность
Дампы регистров AY	Транспонирование (шаг = 14, по одному на регистр) + дельта	Каждый регистр меняется плавно между кадрами
Цветовые рампы / градиенты	1-я производная	Линейная или близкая к линейной прогрессия
Тайловые карты	Транспонирование (шаг = ширина карты) + дельта	Пространственная локальность: соседние тайлы похожи

Тип данных	Лучшее преобразование	Почему
Данные растрового шрифта	Разделение битовых плоскостей, или хранение как 1-бит + RLE	Много нулевых байт в нижних выносных элементах
Позиции частиц	Сортировка по одной оси, затем дельта-кодирование каждой оси	Сортированный порядок максимизирует дельта-сжатие

Ключевая идея: **каждый байт, сэкономленный бесплатным преобразованием, — это байт, для экономии которого не нужен более медленный упаковщик.** Транспонирование + дельта + Pletter 5 (быстрый распаковщик) часто побеждает сырой Exomizer (медленный распаковщик) на структурированных данных. Ты получаешь лучшую степень сжатия и более быструю распаковку.

Практический конвейер

Понимание алгоритмов сжатия полезно. Интеграция их в конвейер сборки необходима.

От ассета к бинарнику

Конвейер: исходный ассет (PNG) -> конвертер (png2scr) -> упаковщик (zx0) -> ассемблер (sjasmplus) -> файл .tap. Упаковщик работает на твоей машине разработки, не на Spectrum. Для ZX0: zx0 screen.scr screen.zx0. Включи результат директивой INCBIN sjasmplus:

```
compressed_screen:
    incbin "assets/screen.zx0"
```

В рантайме распакуй простым вызовом:

```
ld    hl, compressed_screen      ; source: compressed data
ld    de, $4000                  ; destination: screen memory
call dzx0_standard              ; decompress
```

Интеграция в Makefile

Шаг сжатия принадлежит твоему Makefile, а не твоей голове:

```
zx0 $< $@
```

```
demo.tap: main.asm assets/screen.zx0
    sjasmplus main.asm --raw=demo.bin
    bin2tap demo.bin demo.tap
```

Измени исходный PNG, запусти make, и сжатый бинарник перегенерируется автоматически. Никаких ручных шагов, никакой забытой перепаковки.

Пример: загрузочный экран с ZX0

Полный минимальный пример — распакуй загрузочный экран в видеопамять и жди нажатия клавиши:

```
; 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"
```

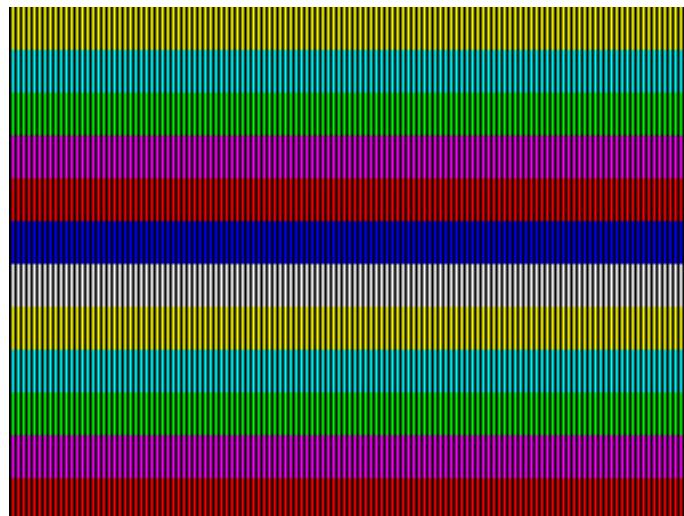


Рис. 30: Демонстрация распаковки ZX0 — сжатый загрузочный экран распаковывается в видеопамять в реальном времени

Используй директиву DISPLAY sjasmplus для вывода информации о размере при ассемблировании. Всегда знай точно, насколько велики твои сжатые данные — разница между ZX0 и Exomizer на одном загрузочном экране может составлять 400 байт, а на 8 сценах это складывается.

Выбор правильного упаковщика

Спрашивай по порядку: (1) sizecoding-интро? ZX0/ZX7 — 69–70 байтный распаковщик не обсуждается. (2) Потоковая передача в реальном времени? LZ4

— ничто другое не достаточно быстро. (3) Однократная загрузка? Exomizer — максимальная степень, скорость не важна. (4) Нужен баланс? ApLib или Pletter 5, оба на Парето-фронтите. (5) Данные полны идентичных серий? Пользовательский RLE. (6) Последовательные кадры анимации? Сначала дельта-кодирование, затем LZ.

Возрождение MegaLZ

В 2017 году Introspec объявил MegaLZ “морально устаревшим”. Два года спустя он сам его воскресил.

Идея: *формат сжатия и реализация распаковщика* — это разделяемые задачи. Формат MegaLZ был хорош — первый Spectrum-упаковщик, использующий оптимальный парсер (LVD, 2005), с гамма-кодами Элиаса и чуть большим окном, чем Pletter 5. Что было плохо — Z80-распаковщик. Introspec написал два новых:

- **Компактный:** 92 байта, ~98 тактов (T-state) на байт
- **Быстрый:** 234 байта, ~63 такта (T-state) на байт — быстрее трёх последовательных LDIR

С этими распаковщиками MegaLZ “убедительно бьёт Pletter 5 и ZX7” по комбинированной метрике степень-плюс-скорость. Урок: не считай упаковщик мёртвым. Формат — это сложная часть. Распаковщик — это Z80-код, а Z80-код всегда можно переписать.

Что значат числа на практике

4К интро: 4 096 байт суммарно. Распаковщик ZX0: ~70 байт. Движок + музыка + эффекты: ~2 400 байт. Остаётся ~1 626 байт на сжатые данные, которые распаковываются в ~3 127 байт сырых ассетов. Megademica от SerzhSoft (1-е место, Revision 2019) сжала тунNELные эффекты, переходы, AY-музыку и быстрые смены сцен ровно в 4 096 байт. Она была номинирована на Outstanding Technical Achievement на Meteoriks.

Потоковая передача в реальном времени: нужно 2 КБ данных на кадр при 50 fps. LZ4 при 34 Т/байт распаковывает 2 048 байт за 69 632 такта (T-state) — почти ровно один кадр (69 888 тактов на 48K). Плотно, но выполнимо с перекрытием распаковки во время бордюра. ApLib потребовал бы 215 040 тактов на те же данные — более трёх кадров. Exomizer — более семи. Для потоковой передачи LZ4 — единственный вариант.

128К мультиценное демо: восемь сцен, каждая с 6 912-байтным загрузочным экраном. Exomizer сжимает каждый до ~3 338 байт; ZX0 до ~3 594 байт. Разница: 256 байт на экран, 2 048 байт на 8 сцен. Когда распаковка происходит при переходе между сценами, медленная распаковка Exomizer незаметна. Экономия 2 КБ — заметна.

256-байтное интро: 70-байтный распаковщик ZX0 оставляет 186 байт на всё. Чаще при таком размере ты пропускаешь LZ и генерируешь данные проце-

дурно с LFSR-генераторами и вызовами калькулятора ПЗУ. Но когда нужны конкретные неалгоритмические данные — цветовая рампа, фрагмент битмапа — ZX0 остаётся инструментом.

Итого: шпаргалка по упаковщикам

Твоя ситуация	Используй	Почему
Однократная загрузка, максимальная степень	Exomizer	48,3% степень, скорость не важна
Универсальное, хороший баланс	ApLib	49,2% степень, ~105 Т/байт
Нужна скорость + приличная степень	Pletter 5	51,5% степень, ~69 Т/байт
Потоковая передача в реальном времени sizecoding-интро (256б-1К)	LZ4	~34 Т/байт, 2+ КБ за кадр
4К интро	ZX0 / ZX7	69-70 байтный распаковщик
Серии идентичных байт	ZX0	Крошечный распаковщик + хорошая степень
Последовательные кадры анимации	RLE (пользовательский)	Распаковщик менее 30 байт
	Дельта + LZ	Использовать межкадровую избыточность

Числа — это ответ. Не мнения, не фольклор, не “я слышал, Exomizer лучший”. Introspec протестировал десять упаковщиков на 1,2 мегабайтах реальных данных Spectrum и опубликовал результаты. Используй его числа. Выбери упаковщик, подходящий под твои ограничения. Затем переходи к сложной части — созданию чего-то, что стоит сжимать.

Попробуй сам

- Сожми загрузочный экран.** Возьми любой .scr файл ZX Spectrum (скачай с zxart.ee или создай свой в Multipaint). Сожми его ZX0 и Exomizer. Сравни размеры. Затем напиши минимальный загрузчик, показанный в этой главе, для распаковки и отображения. Замерь время распаковки, используя тайминг через цвет бордюра из Главы 1.
- Измерь предел потоковой передачи.** Напиши тесный цикл, распаковывающий данные стандартным распаковщиком ZX0, и измерь, сколько байт он может распаковать за кадр. Сравни с распаковщиком LZ4. Проверь числа из таблицы бенчмарка по своим собственным измерениям.

3. **Построй дельта-упаковщик.** Возьми два экрана ZX Spectrum, отличающихся незначительно (сохрани игровой экран, перемести спрайт, сохрани снова). Напиши простой инструмент (на Python или другом языке), который производит дельта-поток: список пар (смещение, новое_значение) для отличающихся байт. Сравни размер дельта-потока с размером полного второго экрана. Затем сожми дельта-поток ZX0 и сравни снова.
4. **Интегрируй сжатие в Makefile.** Настрой проект с Makefile, который автоматически сжимает ассеты как шаг сборки. Измени исходный PNG, запусти make и убедись, что сжатый бинарник перегенерирован и финальный файл .tap обновлён. Это рабочий процесс, который ты будешь использовать в каждом проекте отныне.
5. **Транспонируй и измерь.** Создай 768-байтный файл из 256 троек (X, Y, Z), где X — синусоида, Y — косинусоида, Z — линейная рампа. Измерь энтропию сырого файла. Затем транспонируй его (все значения X, затем все Y, затем все Z) и измерь снова. Примени дельта-кодирование к обеим версиям и сравни. Ты должен увидеть, что транспонированная+дельта версия упадёт ниже 3 бит/байт, тогда как сырая+дельта останется выше 7. Сожми обе версии с ZX0 и сравни реальные размеры — числа энтропии предскажут победителя.
6. **Квадратичная замена.** Сгенерируй 256-байтную таблицу синусов и 256-байтную квадратичную аппроксимацию (подгони $ax^2 + bx + c$ к четверти волны, отзеркаль для полного цикла). Построй графики обеих — они должны быть визуально неразличимы. Теперь вычисли вторую производную каждой. У синуса вторая производная имеет энтропию $\sim 1,5$ бит/байт; у квадратичной — ровно 0. Сожми обе с ZX0. Квадратичная версия меньше, а анимация выглядит так же.

Источники: Introspec “Data Compression for Modern Z80 Coding” (Hype, 2017); Introspec “Compression on the Spectrum: MegaLZ” (Hype, 2019); Break Space NFO (Thesuper, 2016); Einar Saukas, ZX0 (github.com/einar-saukas/ZX0); Ped7g (Peter Helcmanovsky), самомодифицирующийся RLE-распаковщик (предоставлен с разрешения, 2026)

Глава 15: Анатомия двух машин

“Design characterizes realizational, stylistic, ideological integrity.” – Introspec (spke), “For Design” (Hype 2015)

Добро пожаловать в Часть V. Мы создаём игру.

Части I–IV дали тебе инструментарий демосценера: подсчёт тактов, экранные трюки, оптимизацию внутренних циклов, звуковую архитектуру, сжатие. Игра предъявляет другие требования. Тебе нужна полная карта памяти, а не только область экрана. Тебе нужно понимать переключение банков, потому что твои уровни, музыка и данные спрайтов не поместятся в один непрерывный блок. Тебе нужно знать, как два процессора Agon Light 2 общаются друг с другом, потому что твой игровой цикл пересекает эту границу.

Эта глава – аппаратный справочник для всего, что последует. Где Глава 1 дала тебе бюджет кадра, а Глава 2 – раскладку экрана, эта глава даёт всё остальное. Держи её под рукой.

15.1 ZX Spectrum 128K: карта памяти

Оригинальный 48K Spectrum имел простую модель памяти: 16 КБ ПЗУ по адресам \$0000-\$3FFF, 48 КБ ОЗУ по \$4000-\$FFFF. Модель 128K сохраняет эту раскладку, видимую процессору, но прячет систему банков под ней.

128K имеет восемь 16-килобайтных страниц ОЗУ (страницы 0-7, в сумме 128 КБ) и два 16-килобайтных ПЗУ (ROM 0: редактор 128K, ROM 1: 48K BASIC). В каждый момент Z80 видит 64 КБ адресного пространства, разделённого на четыре 16-килобайтных слота:

Диапазон адресов	Содержимое	Примечания
\$0000-\$3FFF	ROM (0 или 1)	Выбирается битом 4 \$7FFD
\$4000-\$7FFF	Страница ОЗУ 5	Всегда страница 5. Здесь живёт экранная память.
\$8000-\$BFFF	Страница ОЗУ 2	Всегда страница 2.

Диапазон адресов	Содержимое	Примечания
\$C000-\$FFFF	Страница ОЗУ N	Переключаемая: любая страница 0-7 через \$7FFD

Страницы 5 и 2 жёстко привязаны к своим слотам. Ты не можешь их заменить. Это означает, что экран (\$4000-\$57FF) всегда доступен, а твой основной код (обычно ORG на \$8000) находится в странице 2, где он не исчезнет при переключении банков.

Верхний 16-килобайтный слот по адресу \$C000-\$FFFF – это гибкий слот. Запиши в порт \$7FFD, и отображённая туда страница изменится.

Порт \$7FFD: переключение банков

Порт \$7FFD управляет конфигурацией памяти на 128К. Он доступен только для записи – ты не можешь его прочитать обратно. Это означает, что ты должен хранить его теневую копию в переменной ОЗУ, если тебе нужно знать текущее состояние.

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

```

Типичная подпрограмма переключения банков:

```

; 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

```

Эти 73 такта (T-state) не бесплатны, но пренебрежимо малы по сравнению с бюджетом кадра 70 000+. Реальная стоимость банкинга – архитектурная, не временная: ты должен проектировать раскладку данных так, чтобы тебе никогда не нужен доступ к двум разным переключаемым страницам одновременно. Музыкальные данные в странице 4, данные уровня в странице 3, спрайтовая графика в странице 6 – но твой музыкальный проигрыватель и рендерер не могут оба работать из \$C000 одновременно.

Теневой экран. Бит 3 \$7FFD выбирает, какую страницу ОЗУ ULA читает для

отображения: страницу 5 (обычная) или страницу 7 (теневая). Это даёт тебе аппаратную двойную буферизацию – рисуй на теневом экране, пока ULA показывает обычный, затем переключи, изменив бит 3. Мы будем активно использовать это в Главе 17 (Скроллинг) и Главе 21 (Полная игра).

Практическая карта памяти для игры на 128К

Вот как реальная игра может распределить свои 128 КБ по восьми страницам:

Страница	Слот	Использование
0	\$C000 (переключаемая)	Данные уровней (карты, определения тайлов)
1	\$C000 (переключаемая)	Набор спрайтовой графики 1
2	\$8000-\$BFFF (фиксированная)	Основной код игры, система сущностей, обработчик прерываний
3	\$C000 (переключаемая)	Набор спрайтовой графики 2, таблицы подстановки
4	\$C000 (переключаемая)	Музыкальные данные (.pt3 паттерны, инструменты)
5	\$4000-\$7FFF (фиксированная)	Основной экран, память атрибутов, системные переменные
6	\$C000 (переключаемая)	Звуковые эффекты, дополнительные данные уровней
7	\$C000 (переключаемая)	Теневой экран (цель двойной буферизации)

Заметь: страница 7 служит двойному назначению. ULA может отображать её как теневой экран, но ты можешь также подключить её к \$C000 и использовать как 16-килобайтную страницу данных, когда не используешь двойную буферизацию. Многие демо это эксплуатируют.

Критическое ограничение: твой обработчик прерываний и основной цикл должны жить в страницах 2 или 5, потому что только эти страницы гарантированно отображены в любое время. Если прерывание сработает, пока страница 4 подключена к \$C000, а твой обработчик прерываний живёт по адресу \$C000, ЦП прыгнет в твои музыкальные данные вместо кода. Результат – крэш, обычно зреющий.

Правило: никогда не помещай критичный по времени код в переключаемую страницу, если ты абсолютно не уверен, какая страница активна, когда этот код выполняется.

15.2 Спорная память: практическая правда

В Главе 1 мы установили, что клоны Pentagon не имеют спорной памяти и что подсчёт тактов надёжен повсюду. Это правда, и Pentagon остаётся стандартом

ZX Spectrum 128K Memory Map

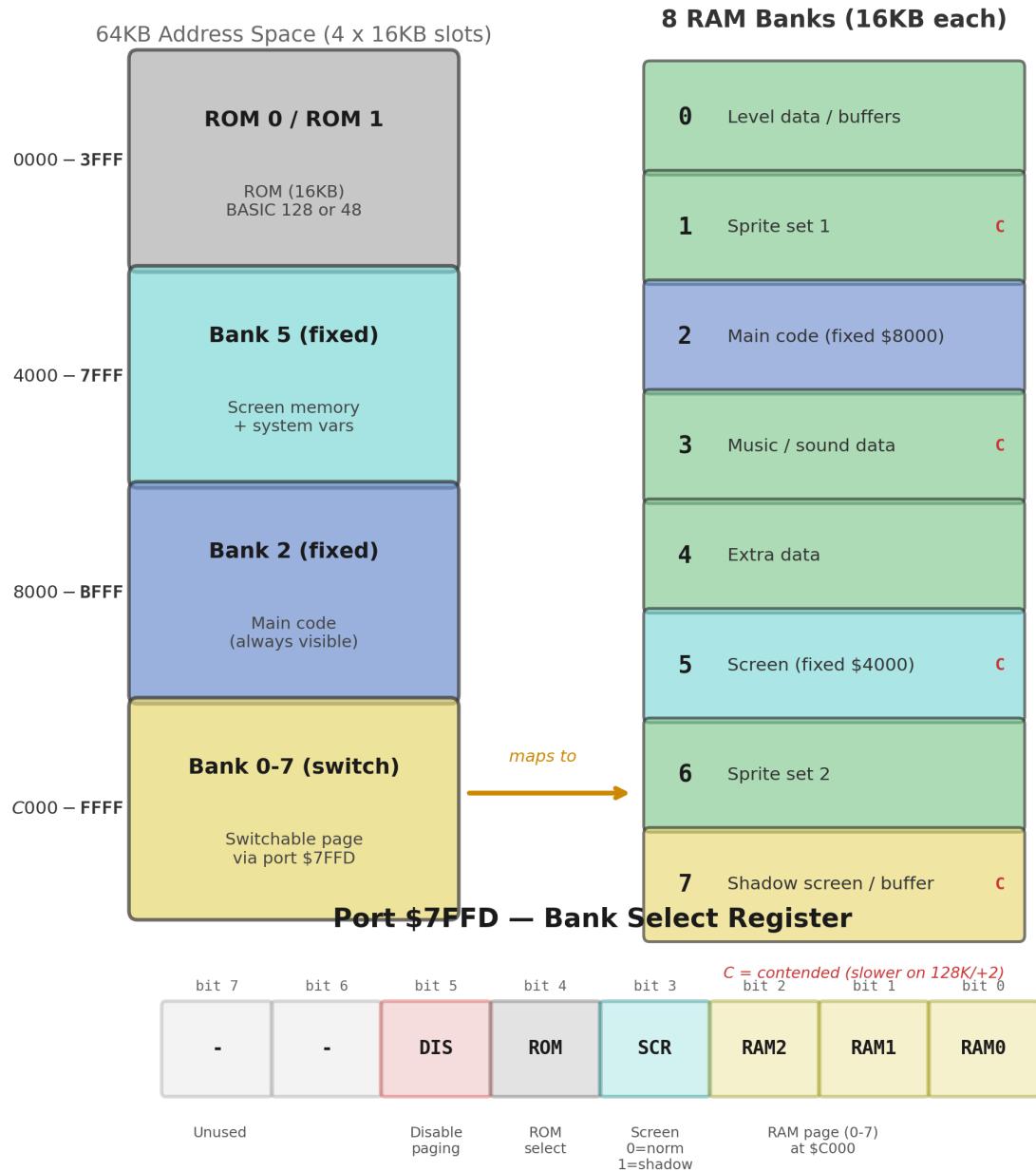


Рис. 31: ZX Spectrum 128K memory map

для потактово точной демосценической работы. Но если ты пишешь игру для релиза, среди твоих игроков будут люди на оригинальном оборудовании Sinclair, моделях Amstrad +2A/+3 и современных FPGA-клонах, эмулирующих оригинальный тайминг. Тебе нужно знать, что делает спорная память и как избежать её худших эффектов.

Introspec исчерпывающе осветил это в своих статьях "GO WEST" на Hype (2015). Вот практическое резюме.

Что подвержено спорности

На оригинальных машинах Sinclair ULA и ЦП разделяют шину памяти. Когда ULA читает данные экрана для отрисовки дисплея (во время 192 активных строк развёртки), любой доступ ЦП к определённым страницам ОЗУ задерживается. ЦП буквально останавливается на дополнительные такты (T-state), пока ULA не завершит своё чтение.

Спорные страницы различаются между моделями:

Модель	Спорные страницы	Всегда быстрые
48K	Только страница 5 (\$4000-\$7FFF)	\$8000-\$FFFF (не спорная)
128K / +2	Страницы 1, 3, 5, 7	Страницы 0, 2, 4, 6
+2A / +2B /	Страницы 4, 5, 6, 7	Страницы 0, 1, 2, 3
+3		

На 48K спорной является только нижняя область ОЗУ в 16 КБ (\$4000-\$7FFF, страница 5) — верхние 32 КБ (\$8000-\$FFFF) не спорные. На 128K паттерн — каждая нечётная страница. На +2A/+3 всё наоборот: верхние страницы спорные.

Это имеет непосредственные практические последствия. На 128K твой основной код по адресу \$8000 (страница 2) находится в неспорной памяти — быстрый. Экран по адресу \$4000 (страница 5) — спорный — записи в экранную память медленнее во время активного отображения. И страница 7 (теневой экран) тоже спорная, что значит, что заполнение двойного буфера на теневом экране медленнее, чем можно ожидать, на оригинальном оборудовании.

Насколько медленнее?

Introspec измерил реальные штрафы:

- **Случайный побайтовый доступ к спорной памяти:** приблизительно **0,92 дополнительных такта (T-state) на байт** в среднем во время активного отображения
- **Стековые операции (PUSH/POP) к спорной памяти:** приблизительно **1,3 дополнительных такта (T-state) на байт** в среднем
- **Во время бордюра: нулевой штраф** — спорность возникает только пока ULA активно рисует строки развёртки

Эта цифра 0,92 означает, что LD A,(HL), которая должна стоить 7 тактов (T-state), будет стоить в среднем около 7,92 тактов, когда HL указывает в спорную память во время активного отображения. PUSH, записывающий два байта в спорную память при 11 тактах, будет стоить около 13,6 тактов вместо этого.

Эти средние скрывают хаотичную реальность: фактический штраф зависит от того, куда в 8-тактовом цикле чтения ULA попадает твой доступ ЦП. Паттерн повторяется каждые 8 тактов: штрафы 6, 5, 4, 3, 2, 1, 0, 0 дополнительных тактов. Ты можешь попасть в любую точку этого цикла, и штраф накапливается с каждым обращением к памяти внутри инструкции. Это делает точный подсчёт тактов на спорных машинах действительно трудным.

Практический ответ

Для разработки игр, а не демо-эффектов, подход прост:

1. **Помести код в неспорную память.** На 128K делай ORG по адресу \$8000 (страница 2) – всегда быстрый.
2. **Пиши на экран во время бордюра, когда возможно.** Верхний и нижний бордюры дают доступ к экранной памяти без спорности. То же касается левого/правого бордюра каждой строки развёртки.
3. **Не беспокойся о точном моделировании спорности.** Заложи 15-20% замедление для кода, обращающегося к экранной памяти во время активного отображения, и проектируй бюджет кадра с этим запасом. Это не потактово точная работа для демо; это разработка игр.
4. **Тестируй на реальном оборудовании или точных эмуляторах.** Fuse правильно эмулирует спорность. Unreal Speccy (режим Pentagon) – нет, по замыслу. ZEsarUX может эмулировать несколько моделей.

Совет Introspec'a из GO WEST сводится к следующему: **спорная память – это проблема переносимости, а не драма.** Если твой код работает на Pentagon, он почти наверняка будет работать и на оригинальном оборудовании – просто чуть медленнее при записи на экран. Места, где спорность реально ломает вещи – это потактово точные растровые эффекты (мультиколор, синхронизация по плавающей шине), а это демо-техники, не игровые.

15.3 Тайминг ULA

ULA генерирует видеосигнал и прерывание ЦП. Понимание её тайминга необходимо для бордюрных эффектов, музыки на прерываниях и синхронизации экрана.

Структура кадра

Полный кадр состоит из строк развёртки. Ширина строки и общее количество строк различаются между моделями:

Машина	Тактов/строка	Строк	Тактов/кадр
ZX Spectrum 48K	224	312	69 888
ZX Spectrum 128K	228	311	70 908
Pentagon 128	224	320	71 680

Заметь более широкую строку 128K (228 против 224 тактов (T-state)). Дополнительные 4 такта на строку – в бордюрной/синхронизационной части, не в

активном дисплее.

Такт-карты: области кадра

Кадр делится на три области. Прерывание срабатывает в начале кадрового гасящего импульса, перед верхним бордюром. Вот временная карта для каждой модели:

Pentagon 128 (71 680 тактов (T-state))

Top border	$80 \text{ lines} \times 224\text{T} = 17,920\text{T}$	No screen reads. No contention.
Active display	$192 \text{ lines} \times 224\text{T} = 43,008\text{T}$	ULA reads screen memory. No contention on Pentagon.
Bottom border	$48 \text{ lines} \times 224\text{T} = 10,752\text{T}$	No screen reads. No contention.
Total: 71,680T		

ZX Spectrum 128K (70 908 тактов (T-state))

Top border	$63 \text{ lines} \times 228\text{T} = 14,364\text{T}$	No screen reads. No contention.
Active display	$192 \text{ lines} \times 228\text{T} = 43,776\text{T}$	ULA reads screen memory. Contention on pages 1,3,5,7.
Bottom border	$56 \text{ lines} \times 228\text{T} = 12,768\text{T}$	No screen reads. No contention.
Total: 70,908T		

ZX Spectrum 48K (69 888 тактов (T-state))

Top border	$64 \text{ lines} \times 224\text{T} = 14,336\text{T}$	No screen reads. No contention.
Active display	$192 \text{ lines} \times 224\text{T} = 43,008\text{T}$	ULA reads screen memory. Contention on all RAM.
Bottom border	$56 \text{ lines} \times 224\text{T} = 12,544\text{T}$	No screen reads. No contention.
Total: 69,888T		

После HALT у тебя есть весь период верхнего бордюра – 17 920 тактов (T-state) на Pentagon, 14 364 на 128K – для работы до того, как луч войдёт в активную область дисплея и начнётся спорность. Вот почему хорошо структурированный код для Spectrum делает записи на экран в начале кадра: ты получаешь доступ к экранной памяти без спорности во время бордюра.

Тайминг строки развёртки

Каждая строка развёртки разбивается на активную часть (где ULA читает данные экрана) и бордюрную/синхронизационную части:

48K и Pentagon (224 такта (T-state) на строку):

```
24T right border
48T horizontal sync + retrace
24T left border
```

128K (228 тактов (T-state) на строку):

```
24T right border
52T horizontal sync + retrace
24T left border
```

Во время 128 активных тактов (T-state) доступ к спорным страницам задерживается (на не-Pentagon машинах). В оставшиеся 96 тактов (или 100 на 128K) – без спорности. Даже во время активного отображения примерно половина каждой строки развёртки свободна от спорности.

Общий и практический бюджет

Общие значения кадра выше – это время между прерываниями. Практический бюджет – такты (T-state), доступные для твоего кода – меньше:

Накладные расходы	Стоимость
HALT + подтверждение прерывания (IM1)	~30 тактов (T-state)
Минимальный обработчик (EI + RET)	~14 тактов (T-state)
Типичный проигрыватель РТЗ (в обработчике)	~3 000-5 000 тактов (T-state)
Обслуживание основного цикла (счётчик кадров, переход HALT)	~20-50 тактов (T-state)

Практические бюджеты с работающим музыкальным проигрывателем:

Машина	Общий	После РТЗ проигрывателя	После проигрывателя + запас на спорность
Pentagon 71	680	~66 000-68 000	~66 000-68 000 (без спорности)
128K	70 908	~65 000-67 000	~55 000-60 000 (записи на экран во время акт. отображения)
48K	69 888	~64 000-66 000	~50 000-55 000 (вся ОЗУ спорная)

Когда эта книга говорит “бюджет кадра ~70 000 тактов (T-state)” – это означает общее значение. При планировании внутренних циклов рассчитывай на практическую цифру – обычно 65 000-68 000 на Pentagon с музыкой.

15.4 Плавающая шина, ULA-снег и баг \$7FFD

Это три аппаратные причуды, проявляющиеся на оригинальном оборудовании Sinclair, но не на большинстве клонов. Ты можешь никогда не столкнуться с ними при разработке игр, но они могут вызвать загадочные баги, если ты не знаешь об их существовании.

Плавающая шина

На оригинальном оборудовании Spectrum чтение из неподключённого порта возвращает данные, которые ULA в данный момент выставляет на шину данных. Во время активного отображения ULA читает экранную память, поэтому чтение из порта \$FF возвращает байт, который ULA сейчас читает.

Демо-кодеры используют это для синхронизации с лучом: читай плавающую шину в тесном цикле, пока не увидишь известное значение из экранной памяти, и ты точно знаешь, где находится луч. Это самый дешёвый метод синхронизации – без расчёта тайминга прерываний.

Игры редко нуждаются в этом, но имей в виду: если твой код читает из порта, который не существует на данном оборудовании, возвращаемое значение непредсказуемо и варьируется между моделями. Плавающая шина *не* эмулируется на Pentagon, Scorpion или ZX Next.

Баг чтения \$7FFD

Порт \$7FFD доступен только для записи. Но на некоторых моделях Spectrum чтение из порта \$7FFD (даже непреднамеренное, через инструкцию, которая случайно выставляет \$7FFD на адресную шину) приводит к тому, что значение плавающей шины записывается в порт. Это вызывает ложное переключение страниц.

Практическая опасность: инструкция Z80 LD A, (nn) выставляет адрес nn на шину во время выполнения. Если nn случайно оказывается \$7FFD и ты читаешь данные, хранящиеся по адресу \$7FFD, чтение памяти может запустить запись в порт на оригинальном оборудовании. Это редкий баг, но реальный. Избегай хранения данных по адресу \$7FFD.

ULA-снег

Если регистр I Z80 (используемый для базы таблицы векторов прерываний IM2) установлен в значение в диапазоне \$40-\$7F, цикл регенерации DRAM во время каждой выборки опкода M1 выставляет адрес в диапазоне \$4000-\$7FFF на адресную шину. Это конфликтует с чтениями экрана ULA и производит визуальный “снег” – случайный шум на дисплее.

Исправление простое: **никогда не устанавливай I в значение между \$40 и \$7F**. Типичная настройка IM2 использует I = \$FE с 257-байтной таблицей идентичных векторов по адресам \$FE00-\$FF00. Это держит I далеко за пределами опасной зоны.

15.5 Различия клонов

Экосистема ZX Spectrum включает десятки клонов, но для современной разработки наиболее важны три: Pentagon 128, Scorpion ZS-256 и ZX Spectrum Next.

Pentagon 128

Pentagon – стандартная платформа для русской демосцены и основная цель демосценических глав этой книги.

Параметр	Pentagon 128	Оригинал 128K
Тактовая частота ЦП	3,5 МГц	3,5 МГц
Тактов (T-state) на кадр	71 680	70 908
Строк развёртки на кадр	320	311
Спорная память	Нет	Страницы 1, 3, 5, 7
Строк бордюра (верх)	80	63
Строк бордюра (низ)	48	56

Дополнительные 772 такта (T-state) на кадр (71 680 против 70 908) приходят от дополнительных строк развёртки. Бордюр распределён по-другому: более высокий верхний бордюр и более короткий нижний. Это влияет на бордюрные эффекты – демо-код, создающий симметричный бордюрный паттерн на 128K, будет слегка асимметричным на Pentagon.

Отсутствие спорной памяти – определяющая черта Pentagon для программистов. Каждая инструкция стоит ровно столько, сколько указано в даташите. Вот почему мы используем тайминг Pentagon на протяжении всей этой книги.

7 МГц турбо-режим. Многие Pentagon-совместимые машины (Pentagon 512, Pentagon 1024, ATM Turbo 2+) предлагают 7 МГц турбо-режим. ЦП работает на двойной скорости, но тайминг ULA остаётся прежним. Это означает, что бюджет кадра удваивается до приблизительно 143 360 тактов (T-state) в турбо-режиме. Подвох: турбо-режим не стандартизирован на всех машинах, и код, опирающийся на него, не будет работать на стоковом Pentagon 128 или любом оборудовании Sinclair.

Для игр турбо-режим – это роскошь, позволяющая запускать более сложную логику или больше спрайтов за кадр. Для демо, нацеленных на правила компо, он обычно запрещён – соревнования указывают “Pentagon 128K, 3,5 МГц”.

Scorpion ZS-256

Scorpion – украинский клон с 256 КБ ОЗУ (16 страниц по 16 КБ) и несколькими аппаратными расширениями.

Параметр	Scorpion ZS-256
ОЗУ	256 КБ (16 страниц)
Банкинг	Расширенный порт \$1FFD для страниц 8-15
Графика	Режим GMX: 320x200, 16 цветов из 256
Спорная память	Нет
Тайминг кадра	Pentagon-совместимый (71 680 тактов (T-state))

Удвоенная ОЗУ полезна для игр: ты получаешь 16 страниц данных вместо 8. Дополнительные страницы доступны через порт \$1FFD, использующий схему, аналогичную \$7FFD, но управляющую дополнительной ОЗУ.

GMX (Graphics Mode Extended) – козырь Scorpion: дисплей 320x200 с 16 цветами из 256-цветной палитры. Это полностью ломает атрибутный дисплей Spectrum, предлагая линейный фреймбуфер, ближе к тому, что можно увидеть на Amiga или PC VGA. Фреймбуфер GMX большой (32 000 байт для 4-битного цвета) и живёт в расширенных страницах ОЗУ.

Немногие игры нацелены на GMX, потому что это ограничивает аудиторию владельцами Scorpion. Но это демонстрирует, на что способно Z80-оборудование, освобождённое от атрибутной сетки ULA.

ZX Spectrum Next

ZX Spectrum Next – современный флагман платформы: машина на FPGA, обратно совместимая с оригинальным Spectrum, но с существенно новым оборудованием.

Параметр	ZX Spectrum Next
ЦП	Z80N (Z80 + новые инструкции) на 3,5 / 7 / 14 / 28 МГц
ОЗУ	1 МБ (расширяемый до 2 МБ), 8 КБ ММУ-страницы
ММУ	8 слотов x 8 КБ = тонкое управление маппингом памяти
Layer 2	256x192 или 320x256, 8-битный цвет (256 цветов)
Тайлмэп	Аппаратный слой тайловой карты, 40x32 или 80x32 тайла
Спрайты	128 аппаратных спрайтов, 16x16, до 12 на строку развёртки
Copper	Сопроцессор для построчных изменений регистров
DMA	zxnDMA для быстрых блочных пересылок
Звук AY	3 x AY-3-8910 (9 каналов) со стерео-панорамированием каждого канала

ММУ Next принципиально отличается от банкинга 128K. Вместо одного переключаемого 16-килобайтного слота Next разделяет всё 64-килобайтное адресное пространство на восемь 8-килобайтных слотов. Каждый слот может быть независимо отображён на любую 8-килобайтную страницу из пула ОЗУ 1-2 МБ. Это означает тонкий контроль:

```
; 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
```

Это намного гибче единственного переключаемого слота 128K. Ты можешь отобразить данные спрайтов в одно 8-килобайтное окно, данные уровня в

другое и музыкальные данные в третье – всё одновременно видимое.

Layer 2 даёт 256-цветный битмап-дисплей без конфликта атрибутов. Это единственное самое значительное улучшение качества жизни для разработчиков игр: больше не нужно тщательное планирование атрибутов, не нужны обходные пути для конфликтов цветов. Просто фреймбуфер, где каждый байт – один пиксель. Цена – память: экран Layer 2 256x192 – это 49 152 байт.

Аппаратные спрайты на Next предоставляют 128 слотов спрайтов, каждый 16x16 пикселей с 8-битным цветом, до 12 на строку развёртки. Атрибуты спрайтов (позиция, паттерн, вращение) устанавливаются через регистры Next и порт \$57. Программный рендеринг не нужен.

Copper – сопроцессор, выполняющий простую программу, синхронизированную с позицией луча. Он может записать в любой регистр Next на любой строке развёртки, обеспечивая построчные смены палитры, смещения скролла и растровые эффекты без потребления тактов (T-state) Z80 – намеренная дань Amiga Copper.

zxnDMA обеспечивает аппаратно-ускоренные блочные пересылки при приблизительно 2 тактах (T-state) на байт – примерно в 10 раз быстрее LDIR. Для заполнения фреймбуфера Layer 2 или пересылки данных спрайтов DMA трансформативен.

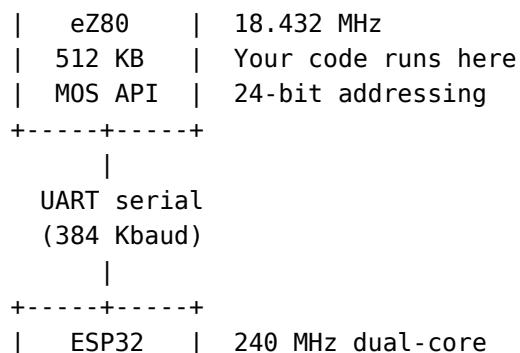
Next по сути – другая машина, которая оказывается обратно совместимой. Интересные ограничения смещаются от “уложусь ли я в бюджет кадра” к “как лучше использовать множественные аппаратные слои”.

15.6 Agon Light 2: другой зверь

Agon Light 2 – вторая платформа для наших глав по разработке игр. Он работает на Zilog eZ80 – прямом потомке Z80 – при 18,432 МГц, с 512 КБ плоской ОЗУ и отдельным сопроцессором ESP32, обрабатывающим видео и аудио. Архитектура принципиально отличается от Spectrum: вместо ЦП, разделяющего шину с фиксированным чипом видеовыхода, Agon использует два независимых процессора, общающихся по последовательной связи.

Двухпроцессорная архитектура

Определяющая характеристика Agon – разделение между **eZ80** (твой ЦП) и **ESP32** (VDP, Video Display Processor):



	FabGL	Video: up to 640x480
	VDP	Audio: waveforms, samples
+-----+		

Это разделение имеет важные последствия:

- Нет общей видеопамяти.** Ты не можешь писать напрямую во фреймбуфер. Каждый пиксель, каждый спрайт, каждая тайловая операция – это команда, отправленная по последовательной связи от eZ80 к ESP32.
- Задержка.** Последовательная связь работает на 384 000 бод. Передача одного байта команды занимает около 26 микросекунд. Сложные операции рисования (заливка прямоугольника, отрисовка битмапа) требуют нескольких байт, и VDP нужно время на их выполнение.
- Асинхронный рендеринг.** VDP обрабатывает команды из буфера. Твой код на eZ80 отправляет команды и продолжает работать. VDP догоняет независимо. Это значит, что у тебя нет плотной связи Spectrum между работой ЦП и выводом на экран – но ты также не можешь точно контролировать, когда пиксели появятся.
- Независимая частота кадров.** VDP рендерит со своей частотой (обычно 60 Гц). Твой игровой цикл eZ80 может работать с любой частотой; VDP покажет то, что он отрисовал последним.

Для программистов Spectrum это совершенно другой подход. Ты переходишь от «я пишу байты в видеопамять и они появляются на следующей строке развёртки» к «я отправляю команды рисования и доверяю VDP отрендерить их когда-нибудь». Плюс — огромное снижение нагрузки ЦП на графику. Минус — меньше контроля.

Модель памяти eZ80: 24-битная плоская

eZ80 имеет 24-битную адресную шину, дающую теоретическое адресное пространство 16 МБ. Agon Light 2 отображает 512 КБ ОЗУ в нижнюю часть этого пространства:

Диапазон адресов	Размер	Содержимое
\$000000-\$07FFFF	512 КБ	ОЗУ
\$080000-\$0FFFFFF	512 КБ	ОЗУ (зеркало, на некоторых платах)
\$A00000-\$FFFFFF	varies	Ввод-вывод, встроенная периферия

Нет банкинга. Нет переключения страниц. Нет спорной памяти. Твой код, данные, буферы, таблицы подстановки — всё живёт в одном плоском, линейно-адресуемом пространстве. После жонглирования 8 страницами Spectrum упрощение ощущается мгновенно.

eZ80 поддерживает два режима работы, определяющих, как он использует адресное пространство.

ADL-режим и Z80-режим

Это самое важное архитектурное различие на Agon, и оно постоянно сбивает с толку новичков.

Z80-режим (также называемый Z80-совместимым режимом) заставляет eZ80 вести себя как классический Z80: 16-битные регистры, 16-битные адреса, 64 КБ адресного пространства. Весь стандартный Z80-код работает без изменений. Верхние 8 бит адреса берутся из регистра MBASE, создавая 64-килобайтное “окно” в 24-битное адресное пространство. Это то, что ты используешь при портировании существующего Z80-кода.

ADL-режим (Address Data Long) – родной режим eZ80: 24-битные регистры, 24-битные адреса, полное 16-мегабайтное адресное пространство. HL, BC, DE, SP, IX/IY – все 24 бита шириной. LD HL,\$123456 загружает 3-байтное значение. PUSH HL помещает 3 байта в стек (не 2). Каждый указатель – 3 байта.

```
; 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 (операционная система Agon) загружает eZ80 в ADL-режиме, и большая часть ПО для Agon остаётся в ADL-режиме. Ключевые отличия от Z80-режима:

Параметр	Z80-режим	ADL-режим
Ширина регистров	16 бит	24 бита
Адресное пространство	64 КБ (через MBASE)	16 МБ (24-битное)
Размер PUSH/POP	2 байта	3 байта
Адреса JP/CALL	16-битные	24-битные
Размер фрейма стека	2 байта на запись	3 байта на запись
Кодирование инструкций	Z80-совместимое	Расширенное (3-байтные адреса)

Ловушка: если ты пишешь код, предполагая 16-битные значения, и запускаешь его в ADL-режиме, вещи ломаются неочевидным образом. PUSH HL помещает 3 байта, не 2, поэтому твои структуры данных на стеке имеют другой размер. JP (HL) прыгает по 24-битному адресу, поэтому таблицы подстановки из 16-битных адресов работать не будут. eZ80 предоставляет инструкции с суффиксами LD.S и LD.L для явного контроля ширины данных, и ты можешь переключаться между режимами с помощью префиксов JP.LIL / JP.SIS, но это быстро становится сложным.

Практическое правило для игр: оставайся в ADL-режиме. Используй 24-битные адреса повсюду. Не пытайся делить код между сборкой для Spectrum и для Agon на уровне исходников – адресация слишком различна. Вместо этого дели алгоритмы и форматы данных, с платформо-специфичными реализациями для доступа к памяти, ввода-вывода и графики.

MOS API: операционная система

MOS (Machine Operating System) предоставляет системные сервисы на Agon: файловый ввод-вывод, ввод с клавиатуры, доступ к таймерам и связь с VDP. Вызовы MOS делаются через RST \$08 с номером функции в регистре A:

```
; 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
```

Ключевые функции MOS для разработки игр:

Функция	Код	Описание
mos_getkey	\$00	Чтение клавиатуры (неблокирующее)
mos_load	\$01	Загрузка файла с SD-карты
mos_save	\$02	Сохранение файла на SD-карту
mos_sysvars	\$08	Получение указателя на системные переменные (счётчик vsync и т.д.)
ffs_fopen	\$0A	Открытие файла
ffs_fclose	\$0B	Закрытие файла
ffs_fread	\$0C	Чтение из файла
mos_getrtc	\$12	Получение часов реального времени

Файловый ввод-вывод на Agon тривиально прост по сравнению со Spectrum. Никакой загрузки с ленты, никаких обёрток esxDOS, никакого TR-DOS: просто открой файл с SD-карты и прочитай его в память. Данные уровней, спрайтовые листы, музыка – загружай их по требованию, без гимнастики с банками.

VDP-команды: разговор с экраном

Вся графика идёт через VDU-команды, отправляемые ESP32 VDP. eZ80 отправляет байты в поток вывода VDU; VDP интерпретирует их как инструкции рисования:

```
; 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
```

Многословно по сравнению с LD (HL),A, но VDP делает рендеринг на ESP32. VDP поддерживает битмап-режимы (до 640x480), до 256 аппаратных спрайтов (каждый до 64x64), аппаратные тайловые карты со скроллингом и аудио (волновые формы, ADSR, сэмплы).

Узкое место – последовательная связь, не ЦП. Сложная сцена с множеством обновлений спрайтов может насытить UART, вызывая визуальный лаг. Минимизируй VDP-команды на кадр: пакетные обновления, используй аппаратный

скроллинг вместо перерисовки тайлов и позволь спрайтовому движку делать тяжёлую работу.

15.7 Сравнение платформ

Давай разложим две машины бок о бок, сфокусировавшись на том, что важно для игрового движка, который мы будем строить в Главах 16-19.

Параметр	ZX Spectrum 128K	Agon Light 2
ЦП	Z80A @ 3,5 МГц	eZ80 @ 18,432 МГц
Тактов (T-state) на кадр	~70 908 (128К, 50 Гц) / 71 680 (Pentagon, 50 Гц)	~307 200 (60 Гц)
ОЗУ	128 КБ (8 x 16 КБ страниц)	512 КБ (плоская)
Адресное пространство	64 КБ (банкованное)	16 МБ (24-битное)
Экранная память	Общая шина, прямая запись	Отдельный VDP, командный
Цвета	15 (8 базовых x яркость, минус перекрытие)	До 64 в стандартных режимах
Разрешение	256x192 (атрибутный цвет на 8x8)	Настраиваемое, до 640x480
Спрайты	Только программные	До 256 аппаратных спрайтов
Скроллинг	Только программный (ручной сдвиг/копирование)	Аппаратные смещения скролла
Звук	AY-3-8910 (3 канала)	ESP32 аудио (многоканальное, волновые формы)
Хранилище	Лента / DivMMC (esxDOS)	SD-карта (FAT32)
Двойная буферизация	Теневой экран (страница 7)	Управляется VDP

Соотношение бюджетов кадров приблизительно 4:1 в пользу Agon. Но графика Agon проходит через узкое место последовательной связи, поэтому сырья скорость ЦП не переводится напрямую в скорость рендеринга. На Spectrum PUSH HL записывает два байта на экран за 11 тактов (T-state). На Agon обновление позиции спрайта требует 6+ байт по каналу 384 Кбод, занимая сотни микросекунд независимо от скорости ЦП.

Spectrum вознаграждает побайтовую оптимизацию. Agon вознаграждает архитектурные решения. Обе платформы вознаграждают внимательное отношение к бюджетам кадров.

15.8 Практика: утилита-инспектор памяти

Давай построим простой инспектор памяти для обеих платформ. Эта утилита показывает область ОЗУ как hex-байты на экране и позволяет навигировать по памяти клавиатурой. Это инструмент, который ты будешь постоянно использовать при разработке.

Версия для Spectrum

Версия для Spectrum пишет прямо в экранную память. Мы показываем 16 строк по 16 байт (256 байт на страницу) с начальным адресом слева.

```
; 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                  ; 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.

```

Ключевой архитектурный момент: мы инспектируем \$C000, потому что это переключаемый слот. Меняя current_bank, мы можем пролистать все 8 страниц ОЗУ, используя подпрограмму bank_switch из раздела 15.1. Сам инспектор живёт по адресу \$8000 (страница 2), защищённый от изменений банкинга.

Версия для Agon

Версия для Agon использует системные вызовы MOS для ввода с клавиатуры и текстовый вывод VDP. Никаких вычислений адреса экрана, никакой обработки атрибутов – просто отправляй текст в VDP.

```

; 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

```

Заметь контраст:

- **Нет переключения банков.** Инспектор Agon может смотреть на любой адрес в 512 КБ напрямую. LD HL,\$070000 – и ты инспектируешь 448 КБ в глубину ОЗУ. Никаких портов, никаких теневых переменных, никакого риска подключить не ту страницу.
- **Нет расчёта адресов экрана.** Текстовый вывод идёт через RST \$10, и VDP обрабатывает позиционирование курсора, рендеринг символов и скроллинг.
- **24-битные директивы данных.** Мы используем dl (define long) для 3-байтных указателей вместо dw (define word).
- **VSync через системные переменные.** MOS предоставляет счётчик sysvar_time, увеличивающийся каждый кадр. Мы крутимся в ожидании его изменения для синхронизации кадров – грубее, чем HALT у Spectrum, но функционально.

Оба инспектора делают одну работу. Версия для Spectrum – больше кода (ты должен обрабатывать всё сам), но даёт полный контроль. Версия для Agon – меньше кода (ОС и VDP обрабатывают отображение), но даёт меньше контроля над тем, как именно выглядит вывод.

Это отражает более широкий опыт разработки на обеих платформах. Spectrum требует больше усилий за меньшее визуальное богатство. Agon требует меньше усилий за большее визуальное богатство. Обе вознаграждают понимание оборудования.

Итого

- **ZX Spectrum 128K** имеет 128 КБ ОЗУ в 8 страницах по 16 КБ. Страницы 2 и 5 фиксированы в адресном пространстве; верхний 16-килобайтный слот по \$C000 переключается через порт \$7FFD. Держи основной код в странице

2 и обработчик прерываний вне переключаемой памяти.

- **Спорная память** замедляет доступ ЦП к определённым страницам ОЗУ во время активного отображения на оригинальном оборудовании Sinclair. Средний штраф: ~0,92 дополнительных такта (T-state) на байт. Клоны Pentagon не имеют спорности. Для разработки игр заложи 15-20% накладных расходов на записи экрана и держи критичный по времени код в неспорных страницах.
- **Тайминг ULA:** прерывание срабатывает в верхней части кадра. Ты получаешь ~14 000 тактов (T-state) свободного от спорности времени до того, как луч войдёт в активную область дисплея. Используй это окно для записей на экран.
- **Порт \$7FFD** доступен только для записи. Храни его теневую копию в ОЗУ. Бит 3 выбирает теневой экран (страница 7) для двойной буферизации. Бит 5 отключает пагинацию навсегда до сброса.
- **Плавающая шина, ULA-снег и баг чтения \$7FFD** – причуды оригинального оборудования Sinclair. Избегай значений регистра I \$40-\$7F. Не храни данные по адресу \$7FFD. Плавающая шина отсутствует на клонах.
- **Pentagon 128:** нет спорной памяти, 71 680 тактов (T-state) на кадр, 320 строк развёртки. Стандарт демосцены. 7 МГц турбо-режим удваивает бюджет кадра на некоторых вариантах.
- **Scorpion ZS-256:** 256 КБ ОЗУ (16 страниц), режим GMX 320x200x16 цветов.
- **ZX Spectrum Next:** 1-2 МБ ОЗУ с 8-килобайтными MMU-страницами, Layer 2 (256-цветный битмап), 128 аппаратных спрайтов, сопроцессор Copper, zxnDMA, тройной AY-звук.
- **Agon Light 2** использует двухпроцессорную архитектуру: eZ80 @ 18,432 МГц для логики, ESP32 для видео/аудио. 512 КБ плоской ОЗУ, 24-битная адресация (ADL-режим), MOS API для системных сервисов, VDP-команды для всей графики.
- **ADL-режим и Z80-режим:** ADL-режим использует 24-битные регистры и адреса. Z80-режим эмулирует классический Z80 с 16-битными адресами через MBASE. Оставайся в ADL-режиме для нового кода Agon.
- **Последовательная связь** между eZ80 и ESP32 – узкое место Agon. Минимизируй трафик VDP-команд на кадр. Используй аппаратные спрайты и тайловые карты для уменьшения количества команд рисования.
- Обе платформы вознаграждают тщательное управление бюджетом кадра. Spectrum даёт тебе ~70 000 тактов (T-state) и требует побайтовой оптимизации. Agon даёт ~307 000 тактов (T-state) (при 60 Гц), но ограничивает графику последовательной связью. Разные ограничения, одна дисциплина.

Источники: 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)

Глава 16: Быстрые спрайты

«Два цвета на ячейку? Ладно. Но эти два цвета будут двигаться.»

Каждой игре нужны объекты, которые движутся. Пули, враги, персонаж игрока, взрывы. На любом железе с блиттером или GPU механика размещения маленького изображения в произвольной позиции на экране решена за тебя. На ZX Spectrum это твоя проблема.

У Spectrum нет аппаратных спрайтов, нет блиттера, нет сопроцессора. Каждый пиксель каждого спрайта размещается твоим Z80-кодом, по одной инструкции за раз, в ту самую экранную память, которую ULA читает 50 раз в секунду. А поскольку раскладка экранной памяти чересстрочная (Глава 2), «на одну строку вниз» означает INC H — если только ты не пересекаешь границу знакоряда, и тогда это означает нечто значительно более уродливое.

В этой главе представлены шесть методов рисования спрайтов на Spectrum, от простейшей XOR-процедуры до скомпилированных спрайтов, которые работают на теоретически максимальной скорости железа. У каждого метода свои компромиссы. Мы также рассмотрим Agon Light 2, где VDP предоставляет аппаратные спрайты и вся проблема сводится к нескольким вызовам API.

Метод 1: XOR-спрайты

Простейший подход

XOR-рисование — это минимально жизнеспособный спрайт. Он не требует данных маски, буфера сохранения фона и шага стирания. Ты рисуешь спрайт, применяя XOR его пиксельных данных к экрану, и стираешь его, применяя XOR тех же данных снова — свойство XOR, что A XOR B XOR B = A, гарантирует восстановление фона.

Вот полная процедура 16x16 XOR-спрайта:

```
; 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

```

Внутренний цикл стоит 134 такта на строку в обычном случае (без пересечения границы знакоряда): две загрузки IX по 19Т каждая, две последовательности XOR-и-запись по 18Т каждая, два INC IX по 10Т, переход к строке (INC H + проверка) — 27Т, и DJNZ — 13Т. На границах знакорядов стоимость возрастает до ~164 тактов (дополнительные инструкции коррекции заменяют выполненный JR). Для 16 строк: примерно 2 200 тактов на отрисовку. Чтобы стереть спрайт, вызови ту же процедуру снова с тем же адресом экрана — XOR отменит сам себя.

Общая стоимость анимации одного XOR-спрайта: ~4 400 тактов на кадр (рисование + стирание).

Когда XOR работает

XOR-спрайты идеальны для:

- **Курсоры.** Мигающий текстовый курсор, перекрестье, выделение. Всё, что лежит поверх статичного фона и не обязано выглядеть красиво.
- **Пули.** Снаряд 2x2 или 4x4, который движется достаточно быстро, чтобы визуальные глюки были незаметны.
- **Отладочные маркеры.** Отображение коллизионных рамок, позиций

сущностей, узлов пути.

Когда XOR не справляется

У XOR две серьёзные проблемы. Во-первых, визуальное качество низкое. Там, где спрайт накладывается на существующие данные экрана, пиксели инвертируются, а не замещаются. Белый спрайт, проходящий поверх белого текста, становится невидимым. Тщательно нарисованный персонаж превращается в мешанину инвертированных пикселей на детальном фоне.

Во-вторых, XOR не даёт тебе контроля над атрибутами. Цвет спрайта — это любая комбинация ink/paper, которая оказалась в ячейках, на которые он попадает. Для пули или курсора это нормально. Для спрайта персонажа — нет.

Несмотря на ограничения, XOR достаточно полезен, чтобы каждый игровой программист имел его в своём арсенале. Двадцать строк кода, ноль дополнительной памяти, и он просто работает.

Метод 2: Маскированные спрайты OR+AND

Отраслевой стандарт

Почти каждая коммерческая игра для Spectrum, выпущенная после 1984 года, использовала эту технику или её близкий вариант. Маскированный спрайт несёт два элемента данных для каждой строки: *маску* и *графику*. Мaska определяет форму спрайта — какие пиксели принадлежат спрайту, а какие прозрачны. Графика определяет внешний вид спрайта — какие из формообразующих пикселей установлены.

Алгоритм рисования для каждого байта:

1. Прочитать байт экрана.
2. Применить AND с маской. Это очищает пиксели, где появится спрайт, оставляя остальной фон нетронутым.
3. Применить OR с графикой. Это впечатывает пиксели спрайта в очищенную область.

Результат: спрайт появляется на экране, а прозрачные области показывают фон сквозь себя. Никаких XOR-артефактов. Никаких инвертированных пикселей. Чистые, профессионально выглядящие спрайты.

Формат данных

Для спрайта 16x16 каждая строка содержит 4 байта: маска-лево, графика-лево, маска-право, графика-право. Байт маски содержит 1 для прозрачных пикселей и 0 для непрозрачных (потому что AND с 1 сохраняет фон, AND с 0 очищает его). Общий размер данных на спрайт: 16 строк x 4 байта = 64 байта.

Внутренний цикл

```
; Draw a 16x16 masked sprite (byte-aligned)
; Input: HL = screen address
```

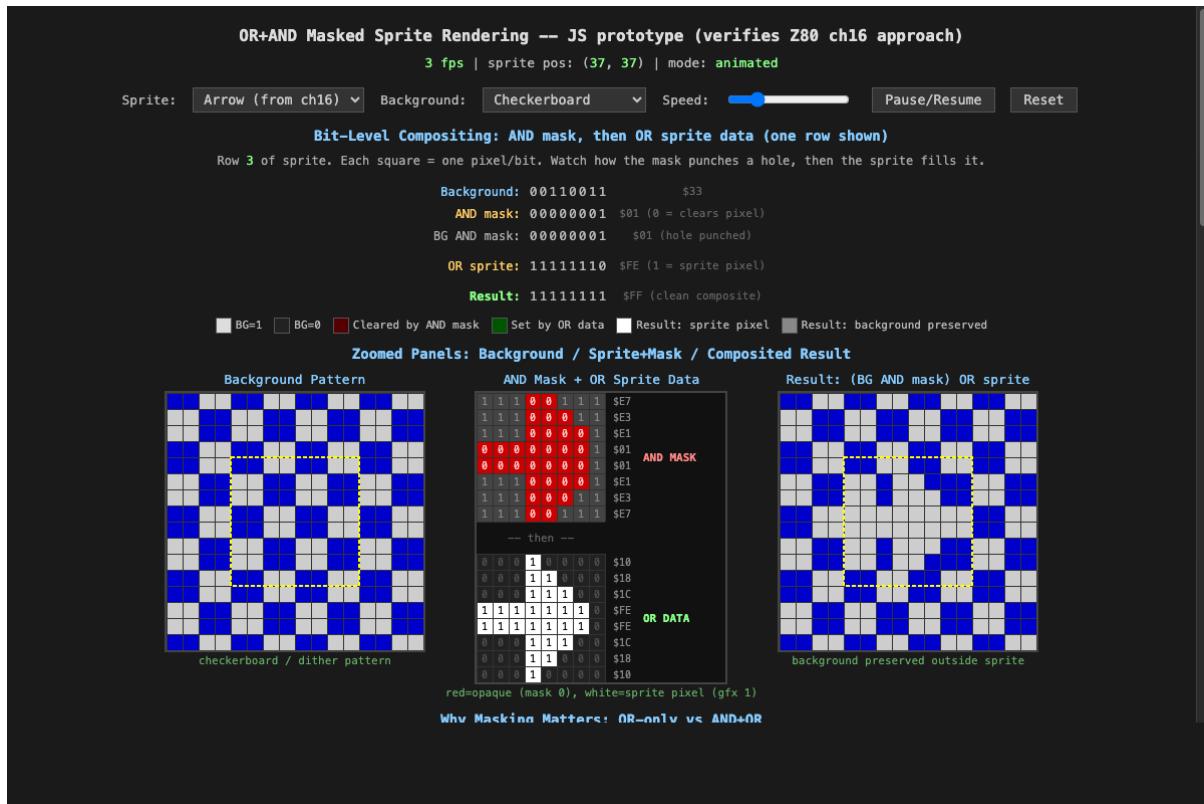


Рис. 32: Композитинг спрайтов AND+OR — маска вырезает прозрачную дыру в фоне (AND), затем данные спрайта заполняют её (OR), сохраняя окружающие пиксели на уровне битов

```

;           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

```

Подсчёт тактов

Подсчитаем обычный случай (без пересечения границы знакоряда). Обрати внимание, что JR NZ выполняется (12T) в обычном случае, потому что пересечение границы — редкость: лишь 1 из 8 строк пересекает границу знакоряда.

Секция	Инструкции	Такты
Левый байт: маска+рисование	ld a,(de) + and (hl) + inc de + ld c,a + ld a,(de) + or c + ld (hl),a + inc de + inc l	52
Правый байт: маска+рисование	Та же последовательность + dec l	52
Переход к строке	inc h + ld a,h + and 7 + jr nz (выполнен)	27
Цикл	djnz	13
Итого на строку		144

Для 16 строк: $16 \times 144 = 2\ 304$ такта (обычный случай). Прибавь накладные расходы на пересечение границ (~ 2 границы в 16-пиксельном спрайте): примерно **2 400 тактов** всего.

Но это только рисование спрайта. Тебе ещё нужно стереть спрайт предыдущего кадра, то есть восстановить фон — мы рассмотрим это в Методе 6 (грязные прямоугольники). Пока заметь, что одно только рисование примерно на 35% дороже XOR, но визуальное качество несравнимо лучше.

Побайтовое выравнивание и проблема сдвига

Процедура выше предполагает, что спрайт начинается на границе байта — то есть координата x кратна 8. На практике игровые персонажи двигаются попиксельно, а не прыжками по 8 пикселей. Если позиция спрайта по x равна 53, он начинается в столбце байт 6, пиксель 5 внутри этого байта. Данные спрайта нужно сдвинуть вправо на 5 бит.

Можно сдвигать во время отрисовки:

```
; 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 /
```

Каждый бит сдвига стоит 24 такта на байт в этом наивном цикле. Для 5-битного сдвига на спрайте шириной 16 (3 байта на строку после сдвига, поскольку спрайт заползает в третий байт), ты получаешь дополнительные $5 \times 24 \times 3 = 360$ тактов на строку — удвоение стоимости отрисовки. Для 8 спрайтов при 25 fps один лишь этот сдвиг съедает примерно 46 000 тактов на кадр — более 60% бюджета кадра.

Вот почему существуют предварительно сдвинутые спрайты.

Метод 3: Предварительно сдвинутые спрайты

Компромисс память-скорость

Идея проста: вместо сдвига данных спрайта во время отрисовки, предвычислить сдвинутые версии спрайта при загрузке (или при ассемблировании) и хранить их рядом с оригиналом. Когда нужно нарисовать спрайт со смещением 3 пикселя внутри байта, ты используешь версию, предварительно сдвинутую на 3 пикселя.

Есть две типичные конфигурации:

4 сдвинутые копии (сдвиг на 0, 2, 4, 6 пикселей). Это даёт горизонтальное разрешение в 2 пикселя. Спрайт привязывается к чётным пиксельным позициям, что часто приемлемо для игровых персонажей. Стоимость памяти: 4x от несдвинутых данных.

8 сдвинутых копий (сдвиг на 0, 1, 2, 3, 4, 5, 6, 7 пикселей). Полное попиксельное горизонтальное позиционирование. Стоимость памяти: 8x от несдвинутых данных. Но каждая сдвинутая версия также шире: спрайт шириной 16 пикселей, сдвинутый на 1-7 бит, заползает в третий столбец байтов, так что каждая сдвинутая копия занимает 3 байта в ширину вместо 2.

Расчёт памяти

Для маскированного спрайта 16x16:

Конфигурация	Байтов на строку	Строк	Копий	Итого
Только без сдвига	4 (маска+графика x 2 байта)	16	1	64
4 сдвига	4	16	4	256
8 сдвигов (3 байта в ширину)	6 (маска+графика x 3 байта)	16	8	768

Для спрайта с 4 кадрами анимации умножь на 4:

Конфигурация	На кадр	4 кадра	8 спрайтов
Без сдвига + сдвиг в рантайме	64	256	2 048
4 предсдвига	256	1 024	8 192
8 предсдвигов	768	3 072	24 576

24 КБ на 8 спрайтов с полным предварительным сдвигом. На 128K Spectrum это полтора банка памяти только под данные спрайтов. На 48K машине это почти половина доступной оперативной памяти. Компромисс жёсткий.

Практический компромисс

Большинство игр используют 4 предварительно сдвинутые копии. Горизонтальное разрешение в 2 пикселя едва заметно в геймплее. Некоторые игры используют 8 копий для персонажа игрока (где плавность движения важнее всего) и 4 копии или даже сдвиг в рантайме для менее важных спрайтов.

Процедура отрисовки для предварительно сдвинутых спрайтов идентична процедуре побайтово выровненного маскированного спрайта — ты просто выбираешь правильный набор предсдвинутых данных перед вызовом:

```
; 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
```

Время отрисовки такое же, как в Методе 2: ~2 300 тактов. Но ты полностью устранил стоимость попиксельного сдвига. Цена заплачена памятью, а не тактами.

Метод 4: Стековые спрайты (метод PUSH)

Самый быстрый вывод на Z80

Мы видели в Главе 3, что PUSH записывает 2 байта за 11 тактов — 5,5 такта на байт, самая быстрая операция записи на Z80. Стековые спрайты используют это для вывода спрайтов: устанавливают SP в нижнюю часть области спрайта на экране, загружают регистровые пары данными спрайта и выполняют PUSH на экран.

Техника требует критической подготовки:

1. **DI** — запретить прерывания. Если прерывание сработает, пока SP указывает на экран, процессор запишет адрес возврата в твои пиксельные данные, повреждая изображение и, вероятно, вызывая сбой.
2. **Сохранить SP** — спрятать настоящий указатель стека с помощью самомодифицирующегося кода (SMC).
3. **Установить SP** на нижний правый угол области спрайта на экране (PUSH работает вниз по адресам).
4. **Загрузить и PUSH** — загрузить данные спрайта в регистровые пары и последовательно выполнить PUSH.
5. **Восстановить SP и EI** — вернуть стек и снова разрешить прерывания.

Внутренний цикл

Для спрайта 16x16 (2 байта в ширину) каждая строка — это один PUSH:

```
; 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.
```

И вот фундаментальная сложность. Метод PUSH — это самая быстрая возможная запись, но чересстрочная раскладка экрана Spectrum означает, что последовательные строки экрана не находятся по последовательным адресам. SP уменьшается линейно, но строки экрана следуют паттерну 010TTSSS LLLCCCCC из Главы 2.

Как заставить работать: предвычисленная цепочка SP

Решение — не полагаться на автоматическое уменьшение SP для навигации по строкам. Вместо этого ты явно устанавливаешь SP для каждой строки:

```
; 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

```

Каждая строка стоит 47 тактов. Для 16 строк: $16 \times 47 = 752$ такта, плюс настройка и завершение (~60 тактов). Итого: примерно **810 тактов**.

Сравни с ~2 300 тактами Метода 2. Стековый спрайт почти в 3 раза быстрее — но он идёт с ограничениями.

Цена

Нет маскирования. PUSH записывает 2 байта безусловно. Он перезаписывает всё, что было на экране. Нет шага AND-с-маской. Спрайт всегда является сплошным прямоугольником — любые «прозрачные» пиксели покажут цвет фона спрайта, а не фон игры. Для спрайтов на однотонном фоне (многие классические игры для Spectrum использовали чёрный) это нормально. Для спрайтов поверх детального фона — нет.

Предвычисленные адреса строк. Тебе нужна таблица экранных адресов для всех 16 строк спрайта, обновляемая при каждом перемещении спрайта. Это стоимость подготовки на каждый кадр — не огромная, но и не бесплатная.

Прерывания отключены. Для 8 спрайтов примерно 6 500 тактов с отключёнными прерываниями. Если музыка работает через IM2, планируй отрисовку спрайтов сразу после HALT.

Данные должны храниться в порядке PUSH. Поскольку PUSH записывает старший байт по адресу (SP-1), а младший по адресу (SP-2), и SP уменьшается *перед* записью, раскладка данных требует внимания. Данные спрайта хранятся в обратном порядке: правый байт строки становится младшим байтом, загружаемым в регистр, левый — старшим байтом.

Когда использовать стековые спрайты

Стековые спрайты — оружие выбора, когда нужна чистая скорость, а фон достаточно прост, чтобы перезапись полными прямоугольниками была приемлема. Аркадные игры с чёрным фоном, оверлеи очков и быстро движущиеся объекты — естественное применение. Метод PUSH также используется для очистки экрана и массового вывода данных (Глава 3), где ограничение «нет маскирования» не имеет значения.

Метод 5: Скомпилированные спрайты

Спрайт — это код

Скомпилированный спрайт доводит философию генерации кода из Главы 3 до логического завершения. Вместо таблицы данных, интерпретируемой про-

цедурой отрисовки, спрайт *сам является* исполняемой процедурой. Каждый видимый пиксельный байт спрайта становится инструкцией LD (HL), н. Прозрачные участки становятся инструкциями INC L или INC H для пропуска. Весь спрайт отрисовывается вызовом CALL.

Как это работает

Рассмотрим простой спрайт 8x8 с некоторыми прозрачными пикселями. В маскированном спрайте ты бы хранил пары маска+графика и выполнял цикл AND/OR. В скомпилированном спрайте ты генерируешь Z80-инструкции при ассемблировании (или при загрузке):

```
; 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  ..#####..
    inc  h                  ; 4 T

    ret                   ; 10 T

; Total: 8 x (10 + 4) + 10 = 122 T-states
; Compare: masked routine for 8x8 = ~600 T-states
```

Это 122 такта для спрайта 8x8. Маскированный подход занимает примерно в 5 раз больше.

Скомпилированный спрайт 16x16

Для более широкого спрайта каждая строка может содержать несколько инструкций LD (HL), n, разделённых INC L:

```
; 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
```

На строку (обычный случай): 32 такта. Для 16 строк с 1-2 пересечениями границ: примерно **570 тактов**.

Компромиссы

Достоинства: - Самый быстрый метод спрайтов с поддержкой маскирования. Ты можешь встроить AND-маскирование в скомпилированные спрайты — каждый байт становится `ld a,(hl) / and mask / or graphic / ld (hl)`, а вместо простого `ld (hl),n`. Даже с маскированием скомпилированный подход избегает накладных расходов на цикл, управление указателями данных и подсчёт строк.
 - Нулевые накладные расходы на цикл. Код полностью линейный. - Прозрачные области ничего не стоят, если они занимают целые байты — ты просто пропускаешь их с помощью `INC L` или `INC H`.

Недостатки: - **Размер кода.** Каждый видимый байт занимает 2 байта кода (`LD (HL), n`). С маскированием (4 инструкции на байт) размер кода примерно утраивается. Полный набор из 8 предсдвинутых скомпилированных спрайтов с 4 кадрами анимации может достигать нескольких килобайт на спрайт.

- **Нет изменения данных в рантайме.** Значения пикселей вкомпилированы в операнды инструкций. Анимация требует отдельной скомпилированной процедуры для каждого кадра. - **Обработка границ вкомпилирована.** Пересечения границ знакорядов находятся на фиксированных позициях, поэтому спрайт должен поддерживать согласованное вертикальное выравнивание, или тебе нужны несколько скомпилированных версий.

Скомпилированные спрайты с маскированием

Для спрайтов, которые должны появляться поверх детального фона, ты компилируешь маску в код:

```
; 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 тактов на байт против 52 тактов на байт в универсальной маскированной процедуре (Метод 2). Экономия достигается за счёт устранения управления указателями, подсчёта циклов и загрузки данных — значения маски и графики являются непосредственными операндами.

Для 16 строк x 2 байта: $16 \times (28 + 28 + 4 + 4 + 4) = 16 \times 68 = \mathbf{1\ 088\ тактов}$. Это примерно вдвое дешевле универсальной маскированной процедуры, с полной поддержкой прозрачности.

Метод	Стоимость отрисовки 16x16	Маскирование	Примечания
XOR-спрайт	~2 200 Т	Нет	Рисование + стирание = ~4 400 Т
Маскированный OR+AND	~2 400 Т	Да	Стандартный подход

Метод	Стоимость отрисовки 16x16	Маскирование	Примечания
Пред-сдвигу-тый маски-рован-ный	~2 400 Т	Да	Нет стоимости сдвига; 4-8х память
Стеко-вый спрайт (PUSH)	~810 Т	Нет	Требуется DI; сплошной пря-моугольник
Скомпи-лирован-ный (без маски)	~570 Т	Нет	Код = спрайт; большой объём
Скомпи-лирован-ный (с маской)	~1 088 Т	Да	Лучшее из обоих; наибольший объём

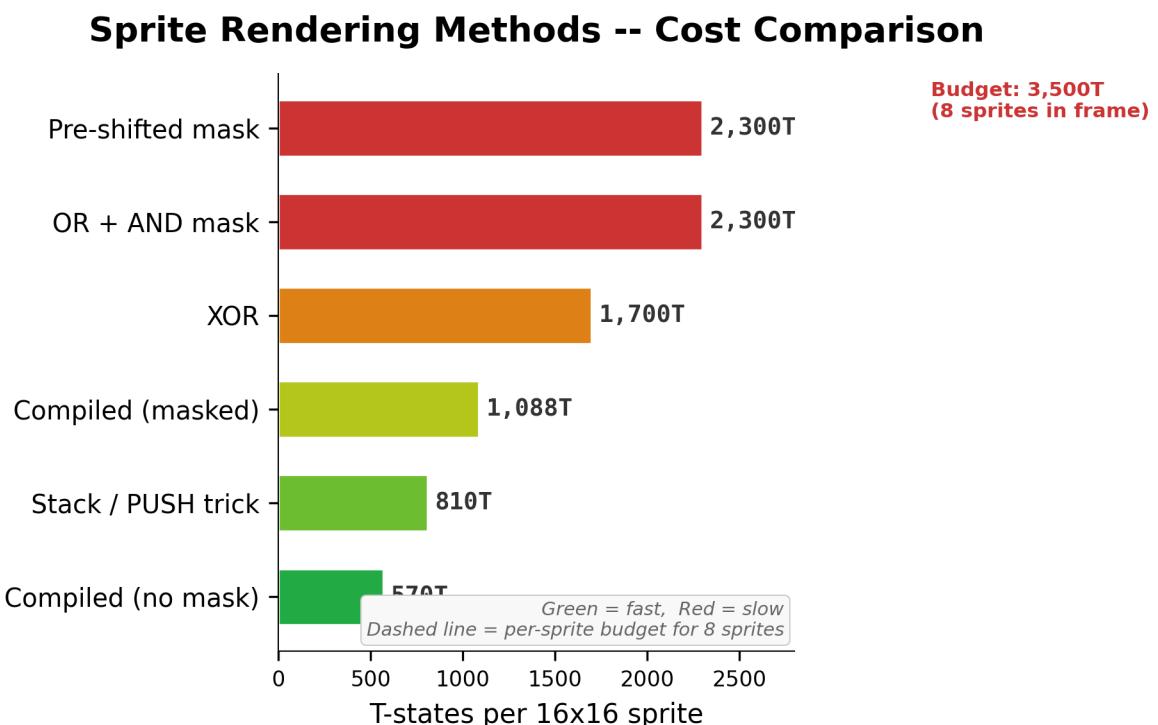


Рис. 33: Sprite rendering methods comparison

Метод 6: Грязные прямоугольники

Проблема фона

Методы 1–5 решают вопрос *размещения* пикселей на экране. Но спрайты двигаются. Каждый кадр спрайт оказывается в новой позиции. Прежде чем нарисовать спрайт на новой позиции, ты должен стереть его со старой — или экран заполнится призрачными послеобразами.

Метод XOR справляется с этим неявно: применяешь XOR на старой позиции для стирания, XOR на новой для рисования. Но для всех остальных методов тебе нужен способ восстановить фон.

Есть три распространённых подхода:

Полная очистка экрана. Стирай пиксельную область каждый кадр (~36 000 тактов с PUSH из Главы 3), затем перерисовывай всё. Осуществимо, но дорого.

Сохранение/восстановление фона. Перед рисованием каждого спрайта сохрани экран под ним. Для стирания скопируй сохранённый буфер обратно. Стоимость $O(\text{размер_спрайта})$ на спрайт, а не $O(\text{размер_экрана})$.

Отслеживание грязных прямоугольников. Усовершенствование: отслеживай, какие прямоугольники были изменены, восстанавливай только их, затем рисуй новые спрайты (сохраняя новый фон по ходу).

Цикл сохранения/восстановления

Практический подход для большинства игр на Spectrum — посправтовое сохранение/восстановление фона. Вот цикл для одного спрайта на кадр:

1. **Восстановить** фон, сохранённый в прошлом кадре (скопировать сохранённый буфер на старую позицию экрана).
2. **Сохранить** фон на новой позиции экрана (скопировать экран в буфер сохранения).
3. **Нарисовать** спрайт на новой позиции.

Порядок важен. Ты восстанавливашь перед сохранением, чтобы не перезаписать новый сохранённый фон устаревшими данными, если спрайты перекрываются.

Процедура сохранения/восстановления

Для спрайта 16x16 (2 байта в ширину, 16 строк) буфер фона составляет 32 байта:

```
; 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

```

Процедура восстановления идентична, только источник и назначение поменяны местами: чтение из буфера, запись на экран. Каждая процедура занимает примерно **1 500 тактов** для 16 строк.

Полный бюджет кадра

Рассчитаем стоимость на кадр для 8 анимированных спрайтов 16x16 с маскированием OR+AND и сохранением/восстановлением фона:

Операция	На спрайт	8 спрайтов
Восстановление фона	~1 500 Т	12 000 Т
Сохранение нового фона	~1 500 Т	12 000 Т
Отрисовка спрайта (маскированная)	~2 400 Т	19 200 Т
Итого	~5 400 Т	~43 200 Т

На Pentagon (71 680 тактов на кадр): 43 200 Т оставляют **28 480 Т** на игровую логику, обработку ввода, музыку и всё остальное. При 25 fps бюджет вдвое больше (два кадра на обновление), давая ~100 000 тактов для неспрайтовой работы. Для игры это комфортно.

Если использовать скомпилированные маскированные спрайты:

Операция	На спрайт	8 спрайтов
Восстановление фона	~1 500 Т	12 000 Т
Сохранение нового фона	~1 500 Т	12 000 Т

Операция	На спрайт	8 спрайтов
Отрисовка спрайта (скомпилированная, маскированная)	~1 088 Т	8 704 Т
Итого	~4 088 Т	~32 704 Т

Это экономит более 10 000 тактов на кадр — ощутимое улучшение, которое даёт больше места для игровой логики или больше спрайтов.

Порядок отрисовки и перекрытие

Когда несколько спрайтов перекрываются, порядок отрисовки важен. Простейший корректный подход:

1. Восстановить все фоны (в обратном порядке отрисовки, чтобы корректно обработать перекрытия).
2. Сохранить все новые фоны.
3. Нарисовать все спрайты.

Восстановление в обратном порядке гарантирует, что когда два спрайта перекрывались в прошлом кадре, буфер сохранения более раннего спрайта (который захватил чистый фон) восстанавливается последним, корректно очищая область перекрытия.

Логика: если спрайт А был нарисован поверх спрайта В, буфер сохранения А содержит пиксели В. Восстановление А первым открывает В, затем восстановление В показывает чистый фон. Восстановление в прямом порядке оставляет артефакты. Многие игры обходят это, предотвращая перекрытие или мирясь с мелкими глюками.

Оптимизация внутренних циклов

Устранение управления указателями

Процедуры выше тратят значительное время на управление указателями: inc de, inc l, dec l и логика пересечения границ DOWN_HL. Несколько оптимизаций могут снизить эти накладные расходы.

Используй LDI вместо ручного копирования. Для операций сохранения/восстановления LDI-цепочка (Глава 3) копирует байт из (HL) в (DE), инкрементирует оба и декрементирует BC — всё за 16 тактов. По сравнению с нашими ручными ld a,(hl) + ld (de),a + inc de + inc l за 24 такта, LDI экономит 8 тактов на байт. Для спрайта 16x16 (32 байта) это 256 тактов экономии на операцию сохранения или восстановления.

```
; Save background using LDI (partial unroll, 2 bytes per row)
; HL = screen address, DE = save buffer
;
save_bg_ldi:
    ld    b, 16          ;  7 Т
```

```

.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

```

Стоимость строки в обычном случае: $16 + 16 + 4 + 4 + 4 + 4 + 7 + 12 + 13 = \mathbf{80 \text{ тактов}}$ (JR NZ выполняется при 12T в обычном случае — нет пересечения границы). Для 16 строк: примерно **1 280 тактов** — стоящее улучшение по сравнению с 1 500 тактами ручного подхода.

Объединение сохранение и рисование. Вместо сохранения-затем-рисования двумя отдельными проходами по области экрана, объедини их в один проход: для каждого байта прочитай экран (сохрани), затем запиши данные спрайта. Это вдвое сокращает количество операций продвижения по строкам и устраняет один полный обход DOWN_HL:

```

; 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

```

Это объединяет сохранение и рисование в один проход. Стоимость на строку (обычный случай): примерно **205 тактов** (JR NZ выполнен при 12T). Для 16 строк: примерно **3 400 тактов** — по сравнению с раздельными сохранением ($\sim 1\ 280$ T) + рисованием ($\sim 2\ 400$ T) = 3 680 тактов. Экономия скромная (~ 280 T на спрайт), но она накапливается по 8 спрайтам.

Для максимальной производительности развёрнутый цикл полностью: никакого DJNZ, явный код для каждой строки с вкомпилированными пересечениями границ на строках 7 и 15. Это устраняет накладные расходы на цикл и проверку границ, снижая итого до примерно **2 780 тактов** ценой ~ 300 байт кода на процедуру спрайта.

Agon Light 2: аппаратные VDP-спрайты

Agon Light 2 использует принципиально другой подход. eZ80 общается с VDP (Video Display Processor) на ESP32, который обрабатывает всю отрисовку спрайтов аппаратно. Процессор загружает битмапы, затем отдаёт команды на позиционирование, показ, скрытие и анимацию спрайтов. VDP компонует спрайты во время собственного прохода рендеринга без нагрузки на процессор за каждый пиксель. Поддерживается до 256 слотов спрайтов.

Последовательность VDU-команд для определения и активации спрайта:

```
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
```

На ассемблере eZ80 эти команды отправляются как последовательности байтов в VDP через RST \$10 (вывод символа MOS). Каждая команда — это последовательность пар ld a, byte : rst \$10.

Перемещение спрайта

После определения перемещение спрайта — это просто команда позиционирования:

```
; 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
```

Стоимость для процессора при перемещении спрайта — лишь отправка ~10 байт по последовательному интерфейсу. При скорости 1 152 000 бод каждый байт занимает примерно 9 микросекунд, так что перемещение одного спрайта занимает примерно 90 микросекунд — около 1 660 тактов при 18,432 МГц. Перемещение 8 спрайтов: ~13 000 тактов. VDP берёт на себя всю пиксельную композицию, прозрачность и управление фоном аппаратно.

Ограничения по строкам развёртки

У VDP есть практическое ограничение на количество пикселей спрайтов на одну горизонтальную строку развёртки. Когда слишком много спрайтов перекрываются на одной линии, некоторые могут мерцать — то же явление, что наблюдается на NES и Master System. Разумная рекомендация — 8-12 спрайтов

16x16 на строку развёртки. Для 8 спрайтов, распределённых по экрану, ты вряд ли столкнёшься с этим ограничением.

Компромисс

Agon устраняет всю проблему отрисовки спрайтов. Никакого сохранения/восстановления, никакого маскирования, никакой навигации по чересстрочному экрану. Цена — абстракция: никаких попиксельных трюков, никаких креативных манипуляций с данными и зависимость от возможностей прошивки VDP. Spectrum заставляет тебя строить всё с нуля. Agon освобождает тебя, чтобы потратить усилия на дизайн игры.

Практика: 8 анимированных спрайтов при 25 fps

Реализация на Spectrum

Наша цель: 8 анимированных спрайтов 16x16 с сохранением/восстановлением фона, работающих при 25 fps (обновление каждые 2 кадра) на ZX Spectrum 128K.

Архитектура:

Каждый спрайт имеет структуру данных:

```
; 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
```

Цикл кадра (каждые 2 VBLANK):

```
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

```

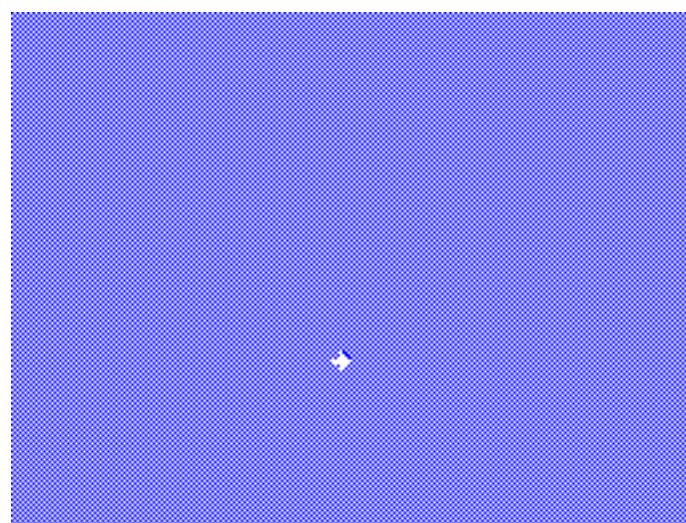


Рис. 34: Маскированный спрайт OR+AND в движении, восемь анимированных спрайтов поверх узорчатого фона

Бюджет тактов:

Фаза	Стоимость
2 x HALT	0 Т (ожидание)
Восстановление 8 фонов	8 x 1 280 = 10 240 Т

Фаза	Стоимость
Обновление 8 позиций	$8 \times 200 = 1\,600$ Т
Сохранение + отрисовка 8 спрайтов	$8 \times 3\,400 = 27\,200$ Т
Накладные расходы на циклы	~2 000 Т
Итого работа со спрайтами	~41 040 Т
Доступно для игровой логики	~102 000 Т

При двухкадровом бюджете в 143 360 тактов ($2 \times 71\,680$ на Pentagon) у нас остаётся примерно 102 000 тактов на игровую логику, ввод и звук. Это щедро — достаточно для ИИ сущностей (Глава 19), обнаружения столкновений с тайлами, воспроизведения музыки и обработки ввода.

Перед рисованием каждого спрайта рассчитай адрес экрана из (x, y) с помощью процедуры из Главы 2 и выбери правильные предсдвинутые данные на основе x AND \$06 (для 4 уровней сдвига). Логика выбора предсдвига из Метода 3 применяется напрямую.

Реализация на Agon

На Agon главный цикл становится тривиально простым: ждёшь VSync, обновляешь позиции, отправляешь команды перемещения VDU 23,27,4 для каждого спрайта и переходишь к игровой логике. Никакого сохранения/восстановления, никакого маскирования, никаких вычислений адресов экрана, никакой навигации по чересстрочным строкам. VDP берёт на себя всё.

Контраст поучителен. На Spectrum рендеринг спрайтов — доминирующая статья расходов: более 40 000 тактов на кадр, где каждый сэкономленный такт во внутреннем цикле напрямую конвертируется в больше спрайтов или больше игровой логики. На Agon рендеринг спрайтов фактически бесплатен с точки зрения процессора, и твои инженерные усилия уходят в дизайн игры, а не перемещение пикселей. Оба подхода по-своему удовлетворяют.

Итого

- **XOR-спрайты** — простейший метод: XOR для рисования, повторный XOR для стирания. ~2 200 тактов для отрисовки спрайта 16x16. Нет маски, не нужно сохранение фона. Визуальное качество низкое (инвертированные пиксели поверх детального фона). Хороши для курсоров, пуль и отладочных маркеров.
- **Маскированные спрайты OR+AND** — отраслевой стандарт. Каждый байт проходит через последовательность AND-с-маской, OR-с-графикой, которая даёт чистую прозрачность. ~2 400 тактов для спрайта 16x16. Именно это использовало большинство коммерческих игр для Spectrum.
- **Предварительно сдвинутые спрайты** устраняют стоимость попиксельного сдвига, храня 4 или 8 предвычисленных сдвинутых копий данных спрайта. Время отрисовки такое же, как у маскированной процедуры. Стоимость памяти масштабируется от 4x (4 сдвига, разрешение 2 пикселя) до

8x (8 сдвигов, полное пиксельное разрешение). Стандартный компромисс память-скорость.

- **Стековые спрайты (метод PUSH)** — самый быстрый необработанный вывод: ~810 тактов для спрайта 16x16. Требуют DI/EI, явного управления SP для каждой строки и создают сплошные прямоугольники (нет маскирования). Лучше всего подходят для игр с простым фоном.
- **Скомпилированные спрайты** превращают спрайт в исполняемый код. Каждый пиксельный байт становится инструкцией LD (HL), п. ~570 тактов без маскирования, ~1 088 тактов с вкомпилированным маскированием. Самый быстрый маскированный метод, ценой большого размера кода. Анимация требует отдельных скомпилированных процедур для каждого кадра.
- **Грязные прямоугольники** с сохранением/восстановлением фона — стандартная техника для анимации спрайтов. Сохрани фон перед рисованием, восстанови его перед рисованием следующего кадра. Восстанавливай в обратном порядке отрисовки для корректной обработки перекрывающихся спрайтов. Объединённый подход сохранения-и-рисования снижает стоимость на спрайт до ~3 400 тактов.
- **8 спрайтов при 25 fps** на Spectrum 128K стоят примерно 41 000 тактов на цикл обновления (каждые 2 кадра), оставляя ~102 000 тактов на игровую логику — комфортный бюджет для настоящей игры.
- **Аппаратные спрайты Agon Light 2** устраняют всю проблему рендеринга. Определи спрайты один раз, перемещай их VDU-командами. Нагрузка на процессор ничтожна. Компромисс — абстракция: ты получаешь производительность, но теряешь возможность попиксельных трюков с данными спрайтов.
- Выбор метода спрайтов зависит от требований твоей игры: сложность фона, количество спрайтов, потребности анимации, доступная память и целевая частота кадров. Большинство игр для Spectrum используют маскирование OR+AND с предсдвигом и грязными прямоугольниками. Демосценовые продукции и игры, критичные к производительности, прибегают к скомпилированным спрайтам или методу PUSH.

Попробуй сам

1. **Реализуй все шесть методов.** Возьми простой дизайн спрайта 8x8 и реализуй версии XOR, маскированную и скомпилированную. Используй тестовую обвязку с цветом бордюра из Главы 1, чтобы сравнить стоимость отрисовки. Разница должна быть отчётливо видна.
2. **Генератор предсдвигов.** Напиши утилиту (на Python, Processing или Z80 ассемблере), которая принимает маскированный спрайт и генерирует 4 предварительно сдвинутые версии. Со храни их в памяти и напиши процедуру отрисовки, которая выбирает правильную версию на основе x-координаты.

3. **Демо сохранения/восстановления фона.** Размести маскированный спрайт на узорчатом фоне (шахматная доска из практики Главы 2). Перемещай спрайт с клавиатуры. Убедись, что фон корректно восстанавливается при движении спрайта. Затем добавь второй спрайт и проверь, что области перекрытия обрабатываются корректно.
4. **Испытание 8 спрайтов.** Реализуй полную систему из 8 спрайтов с сохранением/восстановлением фона и анимацией. Начни с маскированного подхода OR+AND. Измерь бюджет тактов цветами бордюра. Если останется запас, переключись на скомпилированные маскированные спрайты и измерь улучшение.
5. **Сравнение с Agon.** Если у тебя есть Agon Light 2, реализуй ту же анимацию 8 спрайтов с использованием аппаратных VDP-спрайтов. Сравни сложность кода и бюджет процессора, доступный для игровой логики.

Источники: фольклор программирования графики на Spectrum, широко документированный в ZX-сообществе; Глава 3 этой книги для PUSH-трюков и самомодифицирующегося кода; Глава 2 для раскладки экрана и DOWN_HL; документация VDP Agon Light 2 (прошивка Quark, FabGL sprite API); главы по разработке игр из *book-plan.md* для фреймворка из шести методов и практических целей.

Глава 17: Скроллинг

«Экран шириной 256 пикселей. Уровень — 8 000. Каким-то образом игрок должен пройти через него.»

Каждой игре с боковым скроллингом нужно двигать мир. Игрок бежит вправо, фон сдвигается влево. Выглядит просто. На оборудовании с регистром скроллинга — NES, Mega Drive, Agon Light 2 — это *действительно* просто: запиши смещение, и железо сделает остальное. На ZX Spectrum регистра скроллинга нет. Нет никакой аппаратной поддержки. Чтобы прокрутить экран, ты двигаешь байты сам. Все 6 144.

Эта глава разбирает все практические методы скроллинга на Spectrum, от самого дешёвого до самого дорогого: скроллинг атрибутов (768 байт, тривиально), вертикальный пиксельный скроллинг (непросто из-за чересстрочной раскладки экрана из Главы 2), горизонтальный пиксельный скроллинг (дорого — каждый байт в каждой строке нужно сдвинуть) и комбинированный метод, который используют настоящие игры для плавного горизонтального скроллинга в рамках приемлемого бюджета. Мы подсчитаем каждый такт (T-state), построим сравнительные таблицы и покажем, как трюк с теневым экраном на 128K делает всё это без разрывов.

Затем мы посмотрим, как Agon Light 2 решает ту же задачу с помощью аппаратных смещений скроллинга и поддержки тайлмапов — полезное сравнение, которое показывает, что на самом деле означает «один и тот же ISA, но другое железо».

Бюджет

Прежде чем мы напишем хоть одну инструкцию, давай определим, с чем мы работаем.

На Pentagon (модель тайминга, на которую ориентируется большинство демо и игр для Spectrum) один кадр — это **71 680 тактов (T-state)**. На стандартном 48K/128K Spectrum — 69 888. Мы будем использовать цифры Pentagon на протяжении всей главы, но анализ применим к обоим — разница составляет около 2,5%.

Полноэкранный скроллинг означает перемещение данных по всем 6 144 байтам экранной памяти (и, возможно, 768 байтам области атрибутов). Вопрос всегда один и тот же: можем ли мы уложиться в один кадр, и если да, сколько

тактов останется на всё остальное — игровую логику, отрисовку спрайтов, музыку, ввод?

Вот чистая стоимость одного только *прохода* по каждому байту пиксельной области разными методами:

Метод	На байт	6 144 байта	% кадра
ldir	21 Т	129 019 Т	180%
LDI-цепочка	16 Т	98 304 Т	137%
ld a,(hl) + ld (de),a + inc hl + inc de	24 Т	147 456 Т	206%
push (2 байта)	5,5 Т/байт	33 792 Т	47%

Первые три метода не могут переместить всю пиксельную область за один кадр. Даже LDI-цепочки, самый быстрый метод копирования после PUSH, превышают бюджет на 37%. А скроллинг — это не просто копирование: горизонтальный скроллинг требует операции *сдвига* для каждого байта, что увеличивает стоимость на байт.

Вот почему скроллинг на Spectrum — это задача проектирования, а не просто кодирования. Нельзя решить полноэкранный пиксельный скроллинг на 50fps грубой силой. Нужно выбирать метод исходя из того, что может себе позволить твоя игра.

Вертикальный пиксельный скроллинг

Вертикальный скроллинг перемещает содержимое экрана вверх или вниз на одну или несколько строк пикселей. Концептуально это просто: скопировать каждую строку на позицию строки выше (для скроллинга вверх) или ниже (для скроллинга вниз). На линейном фреймбуфере это было бы одно блочное копирование. На Spectrum чересстрочная раскладка экрана (Глава 2) делает задачу значительно интереснее.

Проблема чересстрочности

Вспомни структуру адресов экрана из Главы 2:

Low byte: L L L C C C C C

Где TT = треть (0-2), SSS = строка развёртки внутри знакоряда (0-7), LLL = знакоряд внутри трети (0-7), CCCCC = байт столбца (0-31).

Чтобы прокрутить вверх на один пиксель, нужно скопировать содержимое строки N в строку N-1 для каждой строки от 1 до 191. Адреса источника и приёмника для соседних строк пикселей *не* разделены постоянным смещением. Внутри знакоряда последовательные строки отличаются на \$0100 в старшем байте (просто INC H / DEC H). Но на границах знакоряда — каждую 8-ю строку — соотношение меняется: нужно добавить 32 к L и сбросить биты строки развёртки в H. На границах третей (каждую 64-ю строку) корректировка снова другая.

Алгоритм: сдвиг вверх на один пиксель

Подход, который работает с чересстрочностью, а не против неё, использует структуру раздельных счётчиков из Главы 2. Поддерживай два указателя (источник и приёмник) и продвигай оба, используя естественную иерархию экрана: 3 трети, 8 знакорядов на третью, 8 строк развёртки на знакоряд. Внутри каждого знакоряда перемещение между строками развёртки — это просто INC H / INC D для источника и приёмника. На границах знакорядов нужнобросить биты строки развёртки и добавить 32 к L. На границах третей — добавить 8 к H ибросить L. Внутренний цикл копирует 32 байта на строку с помощью LDIR или LDI-цепочки, а продвижение указателей встроено в структуру внешнего цикла.

Анализ стоимости

Для каждой из 191 копии строки мы должны скопировать 32 байта из источника в приёмник. С использованием LDIR:

- На строку: 32 байта x 21 такт (T-state) - 5 = 667 тактов на LDIR, плюс накладные расходы на управление указателями.
- Управление указателями (сохранение/восстановление источника и приёмника, продвижение строки развёртки): примерно 60 тактов на строку внутри знакоряда, больше на границах.

Итого с LDIR: примерно 143 000 тактов. Это примерно **два полных кадра**. Вертикальный пиксельный скроллинг на одну строку с использованием LDIR не умещается в один кадр.

Можно лучше. Заменим LDIR на LDI-цепочку — 32 инструкции LDI на строку:

- На строку: $32 \times 16 = 512$ тактов на LDI, плюс ~50 тактов на управление указателями.
- Итого: $191 \times 562 = \mathbf{107\ 342\ такта}$. Всё ещё превышает бюджет примерно на 50%.

PUSH-трюк здесь неудобен, потому что нам нужно копировать между двумя несмежными областями с непостоянным соотношением. PUSH записывает по последовательным убывающим адресам, что не соответствует чересстрочной схеме источника/приёмника.

Частичный скроллинг: практический подход

В реальности большинство игр не прокручивают весь экран из 192 строк. Типичная игра резервирует:

- Верхние 2 знакоряда (16 пикселей) под строку состояния — не прокручиваются.
- Нижний 1 знакоряд (8 пикселей) под строку очков — не прокручивается.
- Середина: 21 знакоряд = 168 строк пикселей = область скроллинга.

168 строк вертикального пиксельного скроллинга с LDI-цепочками: $168 \times 562 = \mathbf{94\ 416\ тактов}$, или 132% кадра. Всё ещё слишком много для одного кадра, если ты хочешь оставить время на что-то ещё.

Вот почему чистый вертикальный пиксельный скроллинг по 1 пикселию за кадр — редкость в играх для Spectrum. Распространённые подходы:

1. **Скроллинг на 8 пикселей (один знакоряд):** Перемещаем атрибуты и выровненные по знакорядам пиксельные данные. Это гораздо дешевле, потому что копируешь только 21 знакоряд \times 8 строк развёртки = 168 строк, но можно использовать трюк с блочным копированием: внутри каждой трети знакоряды хранятся непрерывными блоками. Стоимость: около 40 000–50 000 тактов с LDIR. Выполнимо.
2. **Скроллинг на 1 пиксель с помощью счётчика:** Визуально скроллим на 1 пиксель за кадр, комбинируя посимвольный скроллинг (дёшево, каждые 8 кадров) со счётчиком пиксельного смещения (рисуем новое содержимое со смещением внутри 8-пиксельного знакоряда). Мы рассмотрим этот комбинированный подход в разделе горизонтального скроллинга ниже, потому что там он нужен гораздо чаще.
3. **Использование теневого экрана (только 128К):** Рисуем сдвинутое содержимое в задний буфер, затем переключаем. Это устраняет разрывы и позволяет распределить работу по кадрам. Мы рассмотрим это далее в главе.

Скроллинг на 8 пикселей (один знакоряд)

Скроллинг на целый знакоряд драматически дешевле, потому что источник и приёмник связаны простым смещением внутри каждой трети. Знакоряды внутри трети расположены через 32 байта в L. Так что скроллинг на один знакоряд вверх означает копирование из L+32 в L для каждой строки развёртки и каждой трети.

Для скроллинга игровой области на один знакоряд ключевое наблюдение в том, что в пределах одной строки развёртки знакоряды хранятся непрерывно (через 32 байта). Стока развёртки 0 знакорядов 0–7 в трети лежит по адресам \$xx00, \$xx20, \$xx40, ..., \$xxE0. Скроллинг N знакорядов вверх на одну позицию в пределах одной строки развёртки — это, таким образом, одно блочное копирование ($N-1$) \times 32 байт.

Для 20-знакорядной игровой области данные одной строки развёртки — это 20 \times 32 = 640 байт. Скроллинг этой строки развёртки означает копирование 19 \times 32 = 608 байт вперёд на 32. Мы делаем это для каждой из 8 строк развёртки, обрабатывая границы третей отдельно.

Примерная стоимость: 8 строк развёртки \times ~12 700 тактов на строку развёртки (608 байт через LDIR) + обработка границ третей = примерно **105 000 тактов**. Это 146% кадра.

Даже посимвольный скроллинг всей игровой области за один кадр — это впритык. Игры справляются с этим, используя:

- **Скроллинг во время гашения (бордюра).** Верхний и нижний бордюр на Pentagon дают примерно 14 000 тактов свободного времени, когда нет конфликтов.
 - **Разделение на два кадра.** Скроллим верхнюю половину в одном кадре, нижнюю — в следующем. Визуальный результат — 25fps скроллинг с 8-пиксельными прыжками.
 - **Использование теневого экрана** (см. ниже).
-

Горизонтальный пиксельный скроллинг

Горизонтальный скроллинг — хлеб с маслом для игр с боковым скроллингом: мир сдвигается влево или вправо по мере движения игрока. И это самый дорогой тип скроллинга на Spectrum, потому что он требует не просто копирования байтов, а их *сдвига*.

Почему горизонтальный скроллинг дорог

Когда ты прокручиваешь экран влево на один пиксель, каждый байт в каждой строке должен сдвинуть свои биты влево на одну позицию, и бит, выпавший с левого края одного байта, должен стать самым правым битом его левого соседа. Это цепочка вращения с переносом по всем 32 байтам каждой строки.

Инструкция Z80 RL (вращение влево через перенос) — инструмент для этого. Для скроллинга влево каждый пиксель сдвигается на одну позицию влево. Бит 7 — самый левый пиксель в байте, бит 0 — самый правый. Сдвиг влево означает, что бит 7 каждого байта выходит и должен войти в бит 0 байта слева. Флаг переноса связывает соседние байты, поэтому мы обрабатываем строку **справа налево**:

```
; 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)
```

Каждый байт обходится в: $15 (\text{RL} (\text{HL})) + 6 (\text{DEC HL}) = \mathbf{21 \text{ такт (T-state)}}$ на байт. Для 32 байт в строке: $32 \times 21 - 6 = \mathbf{666 \text{ тактов}}$ на строку (последний DEC HL нам не нужен).

На самом деле первому байту нужен OR A (4 Т) для сброса переноса. Итого одна строка стоит: $4 + 32 \times 15 + 31 \times 6 = 4 + 480 + 186 = \mathbf{670 \text{ тактов}}$.

Для 192 строк: $192 \times 670 = \mathbf{128 \, 640 \text{ тактов}}$. Это **179% кадра**.

Полноэкранный горизонтальный пиксельный скроллинг на один пиксель не умещается в один кадр с использованием цепочек RL. И это *только сдвиг* — мы ещё не нарисовали новое содержимое на правом краю.

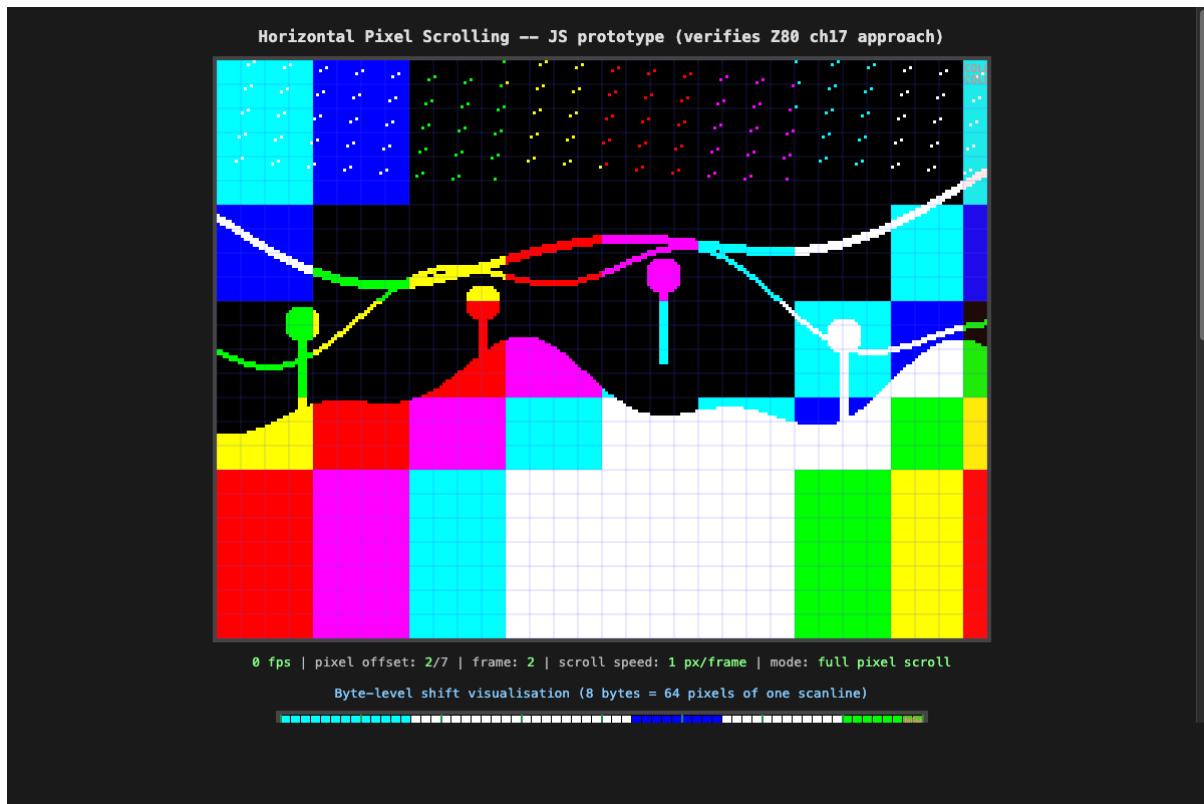


Рис. 35: Прототип горизонтального скроллинга — тайловая игровая область с визуализацией побайтового сдвига, показывающей, как цепочка RL распространяет флаг переноса по смежным байтам

Полный расчёт бюджета

Давай разложим полную стоимость на строку со всеми накладными расходами на навигацию по чересстрочному экрану:

Операция	Тактов на строку
Установить HL на начало строки (или продвинуть от предыдущей)	~15
Установить HL на крайний правый байт: ld a, l : or \$1F : ld l, a	15
Сбросить перенос: or a 32 x rl (hl)	4 480
31 x dec hl (между байтами)	186
Продвинуться к следующей строке (inc h или переход через границу)	4-77
Итого на строку (типично)	~704

Для 192 строк: $192 \times 704 = \mathbf{135\,168\text{ тактов}} = \mathbf{189\% \text{ одного кадра.}}$

Для 168-строчной игровой области: $168 \times 704 = \mathbf{118\,272\text{ такта}} = \mathbf{165\% \text{ одного кадра.}}$

Не существует способа выполнить полноэкранный горизонтальный пиксельный скроллинг на один пиксель за один кадр стандартными методами на Z80 с тактовой частотой 3,5 МГц. Это фундаментальное ограничение, которое определяет каждый метод скроллинга в этой главе.

Можно ли сделать лучше?

Можно подумать, что развёртка циклов или альтернативные режимы адресации помогут. Не помогут. RL (IX+d) стоит 23 такта — *больше*, чем RL (HL) с его 15 Т. Последовательность загрузка-вращение-запись (LD A, (HL) : RLA : LD (HL), A при 18 Т на байт, плюс 6 Т на DEC HL = 24 Т) тоже медленнее. Цепочка RL (HL) : DEC HL при 21 Т/байт — это, по сути, оптимум для горизонтального пиксельного скроллинга на Z80.

Итог: единственный способ сделать горизонтальный скроллинг доступным — уменьшить количество строк или байтов, которые ты прокручиваешь.

Скроллинг атрибутов (посимвольный)

Если пиксельный скроллинг дорог, то скроллинг атрибутов почти бесплатен по сравнению с ним. Скроллинг атрибутов перемещает изображение прыжками по 8 пикселей (одна ячейка атрибутов). Ты перемещаешь только 768 байт области атрибутов и соответствующие выровненные по знакорядам блоки пикселей — или, что чаще, перемещаешь только атрибуты и перерисовываешь игровую область из тайлмапа.

Скроллинг атрибутов с помощью LDIR

Область атрибутов линейна: 32 байта на строку, 24 строки, последовательно от \$5800 до \$5AFF. Скроллинг влево на один столбец символов означает копирование байтов 1-31 на позиции 0-30 в каждой строке, затем запись нового столбца на позицию 31.

Для всей 24-строчной области атрибутов:

```
; 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 такта на полноэкранный скроллинг атрибутов. Это около 24% кадра. Сравни с 135 000+ тактами для пиксельного скроллинга. Скроллинг атрибутов почти в 8 раз дешевле.

Подвох: скроллинг прыгает на 8 пикселей за раз. Визуальный результат — грубый и дёрганый. Для текстовых скроллеров в демо это часто приемлемо — зритель читает текст, а не плавность. Для игры 8-пиксельные прыжки ощущаются ужасно. Вот тут и вступает комбинированный метод.

Комбинированный метод: посимвольный скроллинг + пиксельное смещение

Это техника, которую на самом деле используют большинство игр с боковым скроллингом на Spectrum. Идея проста и мощна:

1. Поддерживай счётчик **пиксельного смещения** от 0 до 7. Каждый кадр увеличивай смещение.
2. Когда смещение достигает 8, сбрось его в 0 и выполнни **посимвольный скроллинг** — дешёвую операцию.
3. Каждый кадр рендири игровую область с применённым текущим пиксельным смещением. Это смещение сдвигает весь дисплей на 0-7 пикселей внутри текущих позиций столбцов символов.

Пиксельное смещение можно применить двумя способами:

Метод А: сдвиг нового столбца. Сдвигай только один столбец пиксельных данных (столбец, входящий в поле зрения) на текущее смещение. Остальная часть экрана рисуется из тайлов с выравниванием по знакорядам. Это работает, когда у тебя есть тайловый рендерер, перерисовывающий из карты.

Метод В: виртуальное смещение в стиле железа. Поддерживай смещение рендеринга, которое контролирует, с какого места внутри каждого знакоряда начинаются данные тайла. Это концептуально похоже на аппаратный регистр скроллинга, но реализовано программно.

Метод А более распространён на практике. Давай разберём его.

Как это работает

Представь, что игровая область имеет ширину 20 символов (160 пикселей) и высоту 20 символов. Данные уровня — это тайлмап, где каждый тайл имеет размер 8x8 пикселей (одна ячейка атрибутов).

Состояние скроллинга состоит из: - `scroll_tile_x`: какой столбец тайлов находится на левом краю экрана (целое число, увеличивается на 1 каждые 8 кадров). - `scroll_pixel_x`: пиксельное смещение внутри текущего тайла (0-7, увеличивается на 1 каждый кадр).

Каждый кадр:

1. **Если `scroll_pixel_x` равен 0:** Перерисовать всю игровую область из тайл-мапа с выравниванием по знакорядам. Это тайловый рендерер, который можно сделать быстрым с помощью LDIR или LDI-цепочек (каждая строка тайла — это 1 байт или несколько байт данных, копируемых по правильному адресу экрана). Стоимость: 20 столбцов x 20 строк x ~100 Т на тайл = ~40 000 Т. Доступно.
2. **Если `scroll_pixel_x` равен 1-7:** Перерисовать игровую область, сдвинутую на `scroll_pixel_x` пикселей. Для большей части игровой области тайлы выровнены по знакорядам и могут быть нарисованы нормально — пиксельное смещение влияет только на **крайний левый и крайний правый видимые столбцы**, где тайл виден частично.

Подожди — это эффективная интерпретация, но она требует тайлового рендерера, который обрезает на субсимвольных границах. Более простой (и более

распространённый) подход:

Простой комбинированный метод

1. Каждые 8 кадров выполняют посимвольный скроллинг (LDIR атрибутов и пиксельных данных влево на один столбец). Стоимость: $\sim 17\ 000$ Т для атрибутов + $\sim 40\ 000$ Т для пиксельных данных = $\sim 57\ 000$ Т. Выполняется раз в 8 кадров.
2. Каждый кадр сдвигает **узкое окно** на 1 пиксель. Это окно шириной всего 1 столбец (32 байта) или 2 столбца (64 байта) — шов между старыми данными и входящим новым столбцом.
3. **Между посимвольными скроллингами** дисплей показывает последнюю позицию после посимвольного скроллинга со смещением 0–7 пикселей, применённым к краевому столбцу. Игрок воспринимает плавный скроллинг по 1 пикселию за кадр.

Вот разбивка стоимости по кадрам:

Операция	Тактов	Частота
Посимвольный скроллинг (вся игровая область)	$\sim 57\ 000$	Каждый 8-й кадр
Пиксельный сдвиг 1–2 краевых столбцов (20 строк x 2 столб. x 21 Т/байт x 8 строк развёртки)	$\sim 6\ 720$	Каждый кадр
Отрисовка нового столбца тайлов на правом краю	$\sim 5\ 000$	Каждый 8-й кадр
Обновление столбца атрибутов	$\sim 1\ 200$	Каждый 8-й кадр

В 7 из 8 кадров: $\sim 6\ 720$ тактов на краевой пиксельный сдвиг. Это менее 10% бюджета кадра. Достаточно места для игровой логики, спрайтов и музыки.

Каждый 8-й кадр: $\sim 6\ 720 + 57\ 000 + 5\ 000 + 1\ 200 = \sim 69\ 920$ тактов. Это 97,5% бюджета кадра. Впритык, но выполнимо — особенно если разбить посимвольный скроллинг на два кадра или использовать теневой экран.

Реализация: пиксельный сдвиг краевого столбца

Ключевая внутренняя подпрограмма сдвигает 1 или 2 столбца пиксельных данных на 1 пиксель. Для 2-столбцовного (16-пиксельного) окна в каждой строке нужно сдвинуть 2 байта:

```
; 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
```

КОМБИНИРОВАННЫЙ МЕТОД: ПОСИМВОЛЬНЫЙ СКРОЛЛИНГ + ПИКСЕЛЬНОЕ СМЕШАНИЕ

```
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)
```

Для 160 строк (20 знакорядов x 8 строк развёртки): $160 \times 40 = \mathbf{6\,400\ тактов}$. С накладными расходами на продвижение указателей (~20 Т на строку) итого около **9 600 тактов** за кадр. Очень доступно.

Конвейер рендеринга

Вот полная покадровая последовательность для комбинированного горизонтального скроллера:

```
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
```

Это скелет реального скроллера для Spectrum. Ключевое наблюдение: плавный скроллинг по 1 пикселью достигается *без сдвига* всего экрана каждый кадр. Дорогой посимвольный скроллинг происходит только раз в 8 кадров, а покадровая работа минимальна.

Скроллинг пиксельных данных на один столбец символов

Посимвольный пиксельный скроллинг (шаг 2 в конвейере выше) сдвигает данные на 8 пикселей влево для каждой строки. Поскольку 8 пикселей = 1 байт, это *побайтовое* копирование, а не побитовое вращение. Каждые 32 байта строки сдвигаются влево на 1 байт: byte[1] переходит в byte[0], byte[2] в byte[1], ..., byte[31] в byte[30], а byte[31] очищается или заполняется новыми данными.

Для одной строки это LDIR на 31 байт:

```
; 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.
```

Для всей игровой области (168 строк): $168 \times 646 = 108\,528$ тактов + накладные расходы на навигацию по строкам.

Лучший подход использует тот факт, что в пределах каждой строки развёртки знакоряда байты идут непрерывно. Для 20 столбцов символов данных одной строки развёртки — это 20 непрерывных байт. Скроллинг этой строки развёртки влево на 1 байт означает LDIR из 19 байт:

```
; 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
```

Для 160 строк: $160 \times 394 = 63\,040$ тактов. Добавим ~20 Т на строку для навигации указателей: $160 \times 414 = \mathbf{66\,240 тактов}$. Это 92% кадра. Выполнимо, но впритык по бюджету «каждый 8-й кадр».

С LDI-цепочками (19 LDI на строку): $19 \times 16 = 304$ Т на строку. Для 160 строк: $160 \times 324 = \mathbf{51\,840 тактов} = 72\%$ кадра. Теперь у нас остаётся 28% на отрисовку нового столбца и обновление атрибутов.

Триок с теневым экраном

ZX Spectrum 128K имеет возможность, которая преображает задачу скроллинга: **два экранных буфера**. Стандартный экран живёт по адресу \$4000 на странице 5 (всегда отображается в \$4000–\$7FFF). Теневой экран живёт по адресу \$C000 на странице 7 (отображается в \$C000–\$FFFF, когда страница 7 подключена).

Порт \$7FFD управляет тем, какой экран отображается:

```
; 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
```

Трюк для скроллинга:

- Кадр N:** Игрок видит стандартный экран (страница 5). Тем временем ты рисуешь содержимое *следующего* кадра со скроллингом в теневой экран (страница 7, по адресу \$C000).
- Кадр N+1:** Переключи отображение на теневой экран. Игрок теперь видит свежеотрисованный кадр без разрывов. Тем временем ты начинаешь рисовать кадр N+2 в теперь скрытый стандартный экран.

Этот подход с двойной буферизацией полностью устраняет разрывы и даёт тебе полный кадр (или больше) на подготовку каждого сдвинутого кадра. Цена — необходимость поддерживать два полных состояния экрана, и каждый «скроллинг» на самом деле представляет собой полную перерисовку игровой области в задний буфер.

```
; 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

```

Стратегия скроллинга с теневым экраном

С двойной буферизацией подход к скроллингу меняется:

Вместо скроллинга активного экрана на месте (что вызывает разрывы и должно завершиться за один кадр), ты **перерисовываешь игровую область из тайлмапа** в задний буфер на новой позиции скроллинга. Это принципиально другой подход. Ты не *перемещаешь* существующие данные экрана — ты *рендеришь заново* из карты.

Это больше работы за кадр (ты перерисовываешь всю игровую область, а не просто сдвигаешь), но у этого есть значительные преимущества:

1. **Нет разрывов.** Игрок никогда не видит наполовину прокрученный экран.
2. **Нет сдвига краевого столбца.** Ты рендеришь каждый тайл с правильным субсимвольным смещением напрямую.
3. **Гибкая скорость скроллинга.** Можно прокручивать на 1, 2 или 3 пикселя за кадр без изменения логики рендеринга.
4. **Более простой код.** Тайловый рендерер проще, чем комбинированный скроллер со сдвигом и копированием.

Стоимость полной перерисовки игровой области из тайлов зависит от твоего тайлового рендерера. При 20 x 20 тайлах, каждый тайл из 8 байт (8 строк развёртки x 1 байт), с использованием LDI-цепочек:

- 400 тайлов x 8 байт x 16 Т на LDI = 51 200 тактов на вывод данных.
- Плюс вычисления адресов тайлов и экрана: ~20 Т на тайл x 400 = 8 000 Т.
- **Итого: ~59 200 тактов = 82% кадра.**

Остаётся 18% (~12 900 тактов) для спрайтов, игровой логики и музыки. Впритык, но рабочий вариант.

Сравнение: методы скроллинга на ZX Spectrum

Метод	Тактов/кадр	% кадра	Визуальное качество	Примечания
Полный пиксельный скроллинг (гориз., 1px)	~135 000	189%	Плавный	Невозможно на 50fps

Метод	Тактов/кадр	% кадра	Визуальное качество	Приме- чания
Полный пиксель- ный скрол- линг (верт., 1px)	~107 000	149%	Плавный	Невоз- можно на 50fps
Только скрол- линг атрибу- тов	~17 000	24%	Дёрганый (прыжки 8px)	Очень дёшево
Комбини- рованный (посимв. + пик- сельный край)	~10 000 средн., ~70 000 пик.	14%/98%	Плавный	Лучший метод для одного буфера
Теневой экран + тайловая перери- совка	~59 000	82%	Плавный, без разрывов	Требует 128K
Посим- вольный скрол- линг (прыжки 8px)	~52 000-66 000	73-92%	Дёрганый	Для скрол- линга тек- ста/ста- туса

Скроллинг вправо (и проблема направления)

Всё описанное выше относится к скроллингу влево (игрок двигается вправо, мир сдвигается влево). А как насчёт скроллинга вправо?

Для скроллинга атрибутов — меняем направление LDIR. Копируем байты 0-30 на позиции 1-31, справа налево. LDIR копирует вперёд (от младших адресов к старшим), поэтому для скроллинга вправо нужен LDDR (копирование назад):

```
; 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
```

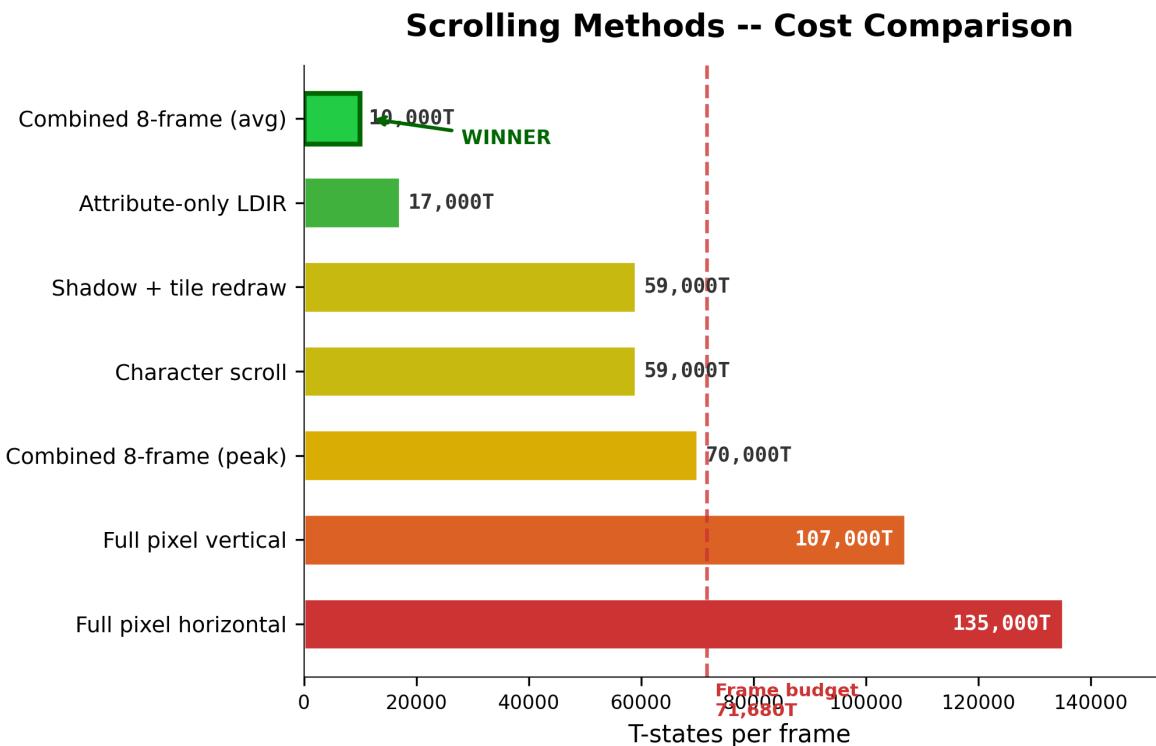


Рис. 36: Scrolling technique cost comparison

```
ret
```

Для побитового пиксельного сдвига скроллинг вправо использует RR (HL) вместо RL (HL), обрабатывая строку слева направо:

```
; 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 ...
```

Стоимость на байт идентична: 21 такт (T-state). Скроллинг вправо стоит столько же, сколько скроллинг влево. Комбинированный метод работает в обоих направлениях с одинаковым бюджетом.

Для двунаправленного скроллинга (игрок может идти и влево, и вправо) нужны две версии подпрограмм посимвольного скроллинга и краевого сдвига, переключаемые в зависимости от направления. Самомодифицирующийся код (SMC) здесь полезен: перед скроллингом пропатчи опкод RL/RR и направление INC/DEC в подпрограмме сдвига. Это позволяет избежать ветвления внутри внутреннего цикла (см. паттерн SMC в Главе 3).

Agon Light 2: аппаратный скроллинг

VDP (Video Display Processor) в Agon Light 2 обрабатывает скроллинг совершенно иначе, чем Spectrum. Там, где программист Spectrum должен двигать байты вручную, Agon предоставляет аппаратную поддержку смещений скроллинга и тайлмапов.

Аппаратные смещения скроллинга

VDP поддерживает смещение окна просмотра для растровых режимов. Задавая регистры смещения скроллинга, ты сдвигаешь всё отображаемое изображение без перемещения пиксельных данных. eZ80 отправляет команду VDP через последовательный канал:

```
; 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
```

Железо применяет это смещение при чтении фреймбуфера для отображения. Пиксельные данные не перемещаются, тактов процессора на сдвиг байтов не тратится, и скроллинг идеально плавный на любой скорости. Стоимость для процессора — лишь накладные расходы на последовательную связь (несколько сотен тактов на последовательность VDU-команд).

Тайлмаповый скроллинг

Тайлмаповый режим VDP обеспечивает встроенный тайловый рендеринг. Ты определяешь набор тайлов (паттерны 8x8 или 16x16 пикселей), строишь массив карты, ссылающийся на индексы тайлов, и железо рендерит карту во время отображения. Скроллинг достигается изменением смещения окна просмотра тайлмапа:

```
; 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 ...
```

Тайлмап автоматически заворачивается. По мере того как окно просмотра прокручивается за край карты, железо оборачивается на начало (или ты можешь обновить краевой столбец новыми индексами тайлов — техника загрузки столбцов через кольцевой буфер).

Загрузка столбцов через кольцевой буфер

Для бесконечно прокручиваемого уровня тайлмап действует как кольцевой буфер. Карта шире экрана как минимум на один столбец. По мере скроллинга игрока вправо:

1. Аппаратное смещение скроллинга продвигается на 1 пиксель за кадр (или с любой нужной скоростью).
2. Когда новый столбец тайлов вот-вот войдёт в поле зрения, eZ80 записывает новые индексы тайлов в столбец, который только что ушёл за левый край.
3. Тайлмап оборачивается, и свежезаписанный столбец появляется справа.

```
; 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
```

Работа процессора за кадр минимальна: запись 20 индексов тайлов через VDP-команды, порядка 2 000–3 000 тактов всего. Остаток кадра доступен для игровой логики. Сравни с 59 000+ тактами Spectrum для тайловой перерисовки при скроллинге. Аппаратный тайлмап Agon даёт примерно 20-кратное снижение нагрузки на процессор для скроллинга.

Сравнение: Spectrum против Agon — скроллинг

Аспект	ZX Spectrum	Agon Light 2
Гранулярность скроллинга	Ограничена программно; 1px возможно, но дорого	1px нативно, нулевая нагрузка на CPU
Нагрузка на CPU за кадр	10 000–135 000 Т	500–3 000 Т
Разрывы	Видимы без двойной буферизации	Нет (VDP обеспечивает синхронизацию)
Смена направления	Требует альтернативных подпрограмм или SMC	Поменять знак смещения
Предел размера карты	Ограничен RAM, нет аппаратной поддержки	Размер тайлмапа ограничен памятью VDP
Цвет на тайл	2 цвета на ячейку 8x8 (атрибут)	Полноцветный, попиксельный

Контраст разительный. То, на что программист Spectrum тратит большую часть бюджета кадра — перемещение пиксельных данных по перемешанной раскладке памяти — Agon решает записью в регистр. Решения в проектировании железа пронизывают все уровни программного обеспечения. Ограничения Spectrum вынудили разработку комбинированного метода скроллинга, тайловых движков и трюков с теневым экраном. Ограничения Agon лежат в другой плоскости (задержки последовательного VDP, накладные расходы на команды для сложных сцен).

Практика: горизонтальный скроллинг уровня

Версия для Spectrum: комбинированный посимвольный + пиксельный скроллинг

Построим горизонтальный скроллер с игровой областью 20x20 символов, который прокручивается плавно со скоростью 1 пиксель за кадр. Данные уровня — это тайлмап, хранящийся в банке памяти.

Вот полная структура:

```
; 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

```



Рис. 37: Горизонтальный пиксельный скроллер, демонстрирующий плавный комбинированный скроллинг знакоместами и пикселями поверх тайловой игровой области

Версия для Agon: аппаратный тайлмаповый скроллинг

Версия для Agon драматически проще. Главный цикл вызывает `vsync`, увеличивает 16-битное смещение скроллинга, отправляет его в VDP через подпрограмму `set_scroll_offset` (несколько вызовов `vdu_write`), и каждые 8 пикселей вызывает `ring_buffer_load` для обновления одного столбца индексов тайлов. Весь скроллинг обходится менее чем в 3 000 тактов за кадр, оставляя 365 000+ тактов для игровой логики, ИИ, физики и рендеринга. Версия для Spectrum — это тщательное упражнение в подсчёте тактов, где каждая техника из Глав 2 и

3 сходится воедино, чтобы достичь того, что Agon делает записью в аппаратный регистр.

Вертикальный + горизонтальный: комбинированный скроллинг

Некоторые игры прокручивают экран одновременно в обоих направлениях. На Spectrum применяют комбинированный метод к обеим осям: посимвольный скроллинг + пикельное смещение (0–7) для каждой. Посимвольный скроллинг по каждому направлению происходит раз в 8 кадров. Совпадение обоих в одном кадре — это вероятность 1/64 (примерно раз в 1,3 секунды) — либо смирись с одним пропущенным кадром, либо разбей работу. Покадровая стоимость краевого сдвига по обеим осям: горизонтальные краевые столбцы (~9 600 Т) + вертикальные краевые строки (~6 400 Т) = ~16 000 Т = 22% кадра. Управляемо.

Советы по оптимизации

1. Используй таблицу подстановки экранных адресов

Предвычисли таблицу из 192 экранных адресов (по одному на строку пикселей) в RAM. Стоимость: 384 байта. Выигрыш: 16-битная подстановка из таблицы (около 30 тактов) заменяет вычисление адреса с перетасовкой битов (91 такт).

2. Прокручивай только видимое

Если спрайты закрывают часть игровой области, можно пропустить скроллинг строк за непрозрачными спрайтами. Отслеживай, какие строки нужно прокручивать, с помощью битовой карты «грязных строк». Эта оптимизация окупается, когда спрайты покрывают значительную часть игровой области.

3. Используй PUSH для посимвольного скроллинга

Для посимвольного скроллинга пиксельных данных (копирование 19 байт влево на строку развёртки) хорошо работает PUSH-триюк. Установи SP на конец данных игровой области для каждой строки развёртки, сделай POP 10 байт, сдвинь содержимое регистров и сделай PUSH обратно со смещением на один байт. Это сложно в настройке, но снижает стоимость на строку развёртки на 30–40%.

4. Разбей посимвольный скроллинг по кадрам

Если посимвольный скроллинг (каждый 8-й кадр) слишком дорог для одного кадра, разбей его: прокрути верхнюю половину игровой области в кадре N, а нижнюю — в кадре N+1. Визуальный артефакт (верхняя половина сдвигается на 1 кадр раньше нижней) практически незаметен при 50fps.

5. Трюки с палитрой и атрибутами

Для скроллинга только атрибутов (без пиксельных данных) рассмотри использование изменений FLASH или BRIGHT для создания иллюзии движения на статичной пиксельной сетке. Вращающийся набор цветов атрибутов на выровненных по символам тайлах может имитировать поток, воду или конвейерные ленты без перемещения пиксельных данных вообще.

Итого

- **Полноэкранный пиксельный скроллинг на ZX Spectrum невозможен на 50fps.** Горизонтальный пиксельный скроллинг стоит $\sim 135\ 000$ тактов для 192 строк (189% бюджета кадра). Вертикальный — $\sim 107\ 000$ тактов (149%). Чересстрочная раскладка экрана усложняет вертикальный скроллинг, а отсутствие цилиндрического двигателя делает горизонтальный скроллинг по природе дорогим.
- **Скроллинг атрибутов дёшев** — $\sim 17\ 000$ тактов (24% кадра), но сдвигает с грубым шагом в 8 пикселей.
- **Комбинированный метод** — это то, что используют настоящие игры: посимвольный скроллинг (каждые 8 кадров) плюс покадровый пиксельный сдвиг 1-2 краевых столбцов. Средняя покадровая стоимость — менее 10 000 тактов. Пик на кадрах посимвольного скроллинга ($\sim 70\ 000$ Т) можно сгладить теневым экраном или разбиением на кадры.
- **Теневой экран** (128K, страница 7) обеспечивает двойную буферизацию без разрывов. Рисуй следующий кадр в задний буфер, затем переключай отображение. Это меняет стратегию скроллинга с «сдвигать существующие данные» на «перерисовывать из тайлмапа», что концептуально проще и устраняет разрывы.
- **Направление горизонтального скроллинга** не меняет стоимость. Скроллинг вправо использует RR (HL) вместо RL (HL), слева направо вместо справа налево, с теми же 21 тактами на байт.
- **Вертикальный пиксельный скроллинг** осложнён чересстрочной раскладкой экрана Spectrum. Перемещение на одну строку пикселей вниз означает навигацию по адресной структуре 010TTSSS LLLCCCCC с различными корректировками указателей на границах знакорядов и третей. Подход с раздельными счётчиками из Главы 2 здесь незаменим.
- **Agon Light 2** предоставляет аппаратные смещения скроллинга и тайловый рендеринг, которые снижают нагрузку на CPU до нескольких VDP-команд за кадр (~ 500 -3 000 тактов). Загрузка столбцов через кольцевой буфер поддерживает тайлмап в актуальном состоянии по мере появления нового ландшафта. То, что программист Spectrum строит за 70 000 тактов, Agon решает записью в регистр.
- **Ключевые техники из предыдущих глав** здесь незаменимы: чересстрочная раскладка экрана и навигация раздельными счётчиками (Глава 2), LDI-цепочки и PUSH-трюки для быстрого перемещения данных (Глава

3), а также самомодифицирующийся код (SMC) для подпрограмм скроллинга с переключением направления (Глава 3).

Источники: Introspec «Ещё раз про DOWN_HL» (Hype, 2020) — навигация по чересстрочному экрану; Introspec «GO WEST Part 1» (Hype, 2015) — стоимость спорной памяти; DenisGrachev «Ringo Render 64x48» (Hype, 2022) — скроллинг со смещением на полсимвола; ZX Spectrum 128K Technical Manual — порт \$7FFD и теневой экран; документация VDP для Agon Light 2 — тайлмапы и команды смещения скроллинга.

Глава 18: Игровой цикл и система сущностей

“Игра — это демо, которое слушает.”

Каждый эффект для демо, который мы строили до сих пор, работает в замкнутом цикле: вычислить, отрисовать, повторить. Зритель наблюдает. Коду всё равно, есть ли кто-то в комнате. Игра нарушает этот контракт. Игра *реагирует*. Игрок нажимает клавишу — и что-то должно измениться немедленно, надёжно, в пределах того же бюджета кадра, который мы считаем с Главы 1.

Эта глава о построении архитектуры, которая делает игру возможной на ZX Spectrum и Agon Light 2. Не рендеринг (спрайты были в Главе 16, скроллинг в Главе 17) и не физика (столкновения и ИИ будут в Главе 19). Эта глава — скелет: главный цикл, который всем управляет, конечный автомат, организующий переход от титульного экрана к геймплею и к экрану проигрыша, система ввода,читывающая намерения игрока, и система сущностей, управляющая каждым объектом в игровом мире.

К концу главы у тебя будет работающий каркас игры с 16 активными сущностями — игрок, восемь врагов и семь пуль — укладывающейся в бюджет кадра на обеих платформах.

18.1 Главный цикл

Каждая игра на ZX Spectrum следует одному фундаментальному ритму:

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

Это игровой цикл. Он не сложен. Его сила в том, что он выполняется пятьдесят раз в секунду, каждую секунду, и всё, что испытывает игрок, возникает из этого цикла.

Вот минимальная реализация:

```
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

```

Инструкция HALT — это сердцебиение. Когда процессор выполняет HALT, он останавливается и ждёт следующего маскируемого прерывания. На Spectrum ULA генерирует это прерывание в начале каждого кадра — раз в 1/50 секунды. Процессор просыпается, обработчик IM1 по адресу \$0038 выполняется (на стандартной ROM это просто инкремент счётчика кадров), а затем выполнение продолжается с инструкции после HALT. Код твоего главного цикла отрабатывает, выполняет свою работу и снова попадает на HALT, чтобы дождаться следующего кадра.

Это даёт тебе ровно один кадр тактов на всё. Если работа завершается раньше, процессор пристаивает внутри HALT до следующего прерывания — никаких потерь, никакого дрейфа, идеальная синхронизация. Если работа занимает слишком много времени и прерывание срабатывает до того, как ты дойдёшь до HALT, ты пропускаешь кадр. Цикл всё равно работает (следующий HALT поймает очередное прерывание), но игра падает до 25 fps на этом кадре. Пропускаешь регулярно — получаешь стабильные 25 fps. Пропускаешь сильно — 16.7 fps (каждый третий кадр). Игрок это замечает.

Бюджет кадра: повторение

Мы установили эти числа в Главе 1, но их стоит повторить в контексте игры:

Машина	Тактов на кадр	Практический бюджет
ZX Spectrum 48K	69 888	~62 000 (после накладных расходов на прерывание)
ZX Spectrum 128K	70 908	~63 000
Pentagon 128	71 680	~64 000
Agon Light 2	~368 640	~360 000

“Практический бюджет” учитывает обработчик прерывания, саму инструкцию HALT и накладные расходы на тайминг бордюра. На Spectrum у тебя примерно 64 000 тактов полезного времени на кадр. На Agon — более чем в пять раз больше.

Как типичная игра тратит эти 64 000 тактов? Вот реалистичная разбивка для платформера на Spectrum:

Подсистема	Тактов	% бюджета
Чтение ввода	~500	0.8%
Обновление сущностей (16 сущностей)	~8 000	12.5%
Обнаружение столкновений	~4 000	6.3%
Проигрыватель музыки (PTZ)	~5 000	7.8%
Рендеринг спрайтов (8 видимых)	~24 000	37.5%
Обновление фона/скроллинга	~12 000	18.8%
Разное (HUD, состояние)	~3 000	4.7%
Оставшийся запас	~7 500	11.7%

Эти 11.7% запаса — твоя зона безопасности. Заходишь в неё — начинаешь терять кадры на сложных сценах. Техника профилирования цветом бордюра из Главы 1 — красный для спрайтов, синий для музыки, зелёный для логики — это способ мониторить этот бюджет в процессе разработки. Используй его постоянно.

На Agon та же игровая логика занимает малую долю бюджета. Обновление сущностей, обнаружение столкновений и чтение ввода могут потребовать 15 000 тактов суммарно — около 4% бюджета кадра Agon. VDP обрабатывает рендеринг спрайтов на сопроцессоре ESP32, так что затраты на спрайты со стороны процессора сводятся к накладным расходам на VDU-команды. У тебя огромный запас для более сложного ИИ, большего числа сущностей или просто меньшего стресса.

18.2 Конечный автомат игры

Игра — это не один цикл, а несколько. У титульного экрана свой цикл (анимация логотипа, ожидание нажатия клавиши). У меню свой цикл (подсветка пунктов, чтение ввода). Игровой цикл — то, что мы описали выше. Экран паузы замораживает игровой цикл и запускает более простой. У экрана “Game Over” ещё один.

Самый чистый способ организовать всё это — **конечный автомат**: переменная, хранящая текущее состояние игры, и таблица адресов обработчиков — по одному на состояние.

Определения состояний

```
; 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
```

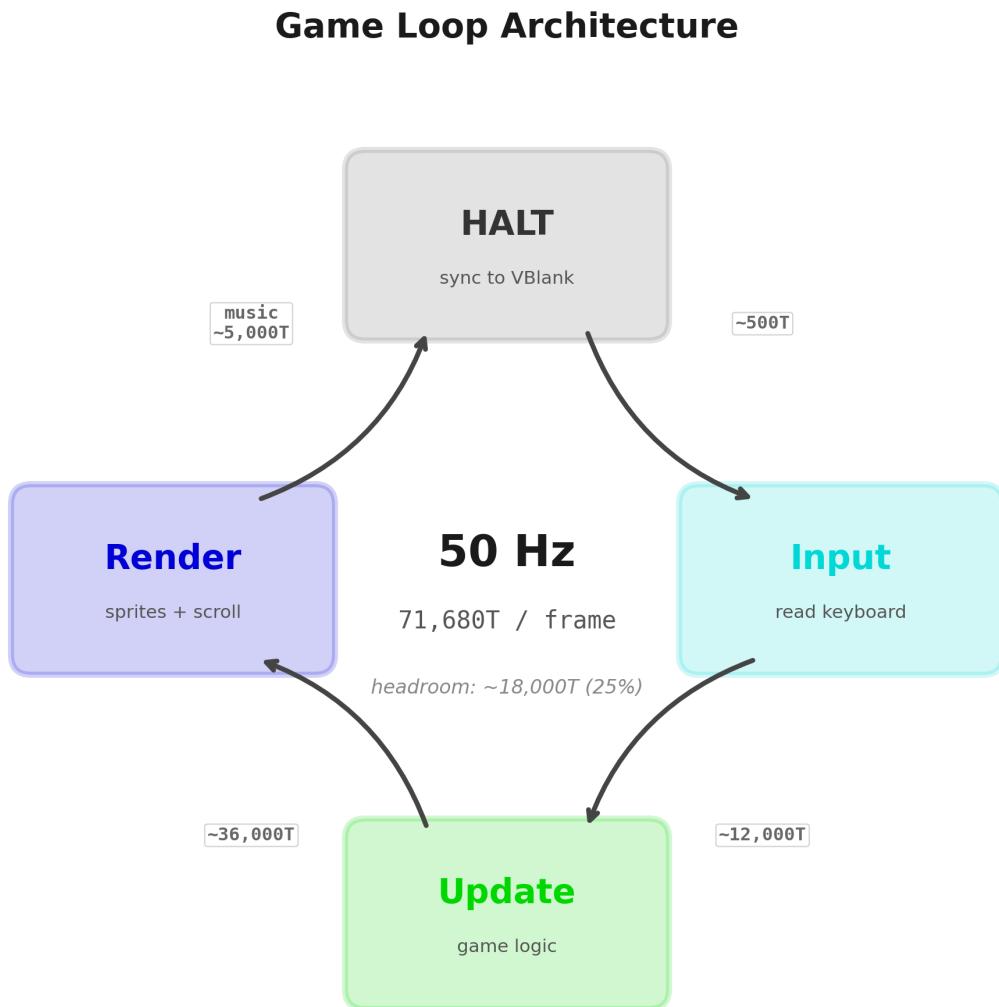


Рис. 38: Game loop architecture

Таблица переходов

```
; 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
```

Диспетчер

Главный цикл становится диспетчером, который читает текущее состояние и переходит к соответствующему обработчику:

```
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
```

Инструкция JP (HL) — ключевая. Она переходит не по адресу, хранящемуся в HL, а по адресу, находящемуся в HL. Это косвенный переход Z80, и он стоит всего 4 такта. Весь диспетч — загрузка переменной состояния, вычисление смещения в таблице, чтение адреса обработчика и переход — занимает 73 такта. Это ничтожно мало: около 0.1% бюджета кадра.

Каждый обработчик выполняет свою логику и затем возвращается к main_loop:

```
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

```

Почему не цепочка сравнений?

Может возникнуть соблазн написать диспетчер так:

```

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
; ...

```

Это работает, но имеет две проблемы. Во-первых, стоимость растёт линейно: каждое дополнительное состояние добавляет СР (7Т) и JP Z (10Т), так что в худшем случае — 17Т на состояние. При 5 состояниях игровое состояние (самый частый случай) может потребовать 51Т на достижение, если оно третье в цепочке сравнений. Таблица переходов стоит 73Т независимо от активного состояния — это O(1), а не O(n).

Во-вторых, что важнее, таблица переходов масштабируется чисто. Добавление шестого состояния (скажем, STATE_SHOP) означает добавление одной записи DW в таблицу и одного определения константы. Код диспетчера не меняется вообще. При цепочке сравнений ты добавляешь больше инструкций в сам диспетчер, и порядок начинает влиять на производительность. Табличный подход и быстрее в типичном случае, и чище в поддержке.

Переходы между состояниями

Переходы между состояниями выполняются записью нового значения в game_state. Обычно при этом вызывается процедура инициализации нового состояния:

```
; 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
```

Делай переходы явными и централизованными. Частый баг в играх для Z80 — переход между состояниями, который забывает инициализировать данные нового состояния: экран проигрыша показывает мусор, потому что никто не очистил экран и не сбросил счётчик анимации. Каждое состояние должно иметь процедуру init_, которую переход вызывает.

18.3 Ввод: чтение игрока

Клавиатура ZX Spectrum

Клавиатура Spectrum читается через порт \$FE. Клавиатура подключена как матрица из 8 полурядов, каждый выбирается установкой бита в ноль в старшем байте адреса порта. Чтение порта \$FE с определённым старшим байтом возвращает состояние этого полуряда: 5 бит, по одному на клавишу, где 0 означает “нажата”, а 1 — “не нажата”.

Карта полурядов:

Старший байт	Клавиши (бит 0 – бит 4)
\$FE (бит 0 = 0)	SHIFT, Z, X, C, V
\$FD (бит 1 = 0)	A, S, D, F, G
\$FB (бит 2 = 0)	Q, W, E, R, T
\$F7 (бит 3 = 0)	1, 2, 3, 4, 5
\$EF (бит 4 = 0)	0, 9, 8, 7, 6

Старший байт	Клавиши (бит 0 – бит 4)
\$DF (бит 5 = 0)	P, O, I, U, Y
\$BF (бит 6 = 0)	ENTER, L, K, J, H
\$7F (бит 7 = 0)	SPACE, SYMSHIFT, M, N, B

Стандартные игровые управление — Q/A/O/P для вверх/вниз/влево/вправо и SPACE для огня — охватывают три полуряда. Вот процедура, которая их считывает и упаковывает результат в один байт:

```
; 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, 0=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  0 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)

```

При примерно 220 тактах в худшем случае чтение ввода тривиально в бюджете кадра. Даже на Spectrum ты можешь позволить себе читать клавиатуру десять раз за кадр и едва это заметить.

Джойстик Kempston

Интерфейс Kempston ещё проще. Одно чтение порта возвращает все пять направлений плюс огонь:

```

; 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

```

Обрати внимание на удобное совпадение: расположение бит Kempston совпадает с нашими определениями INPUT_*. Это не случайность — интерфейс Kempston был разработан с учётом этого стандарта, и большинство игр для Spectrum используют тот же порядок бит. Если ты поддерживаешь и клавиатуру, и джойстик, можно объединить результаты через OR:

```

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

```

Теперь остальной код проверяет только input_flags и не заботится о том, пришёл ли ввод с клавиатуры или джойстика.

Детектирование фронтов: нажатие vs удержание

Для некоторых действий — выстрел, открытие меню — нужно реагировать на событие *нажатия*, а не на состояние удержания. Если проверять bit INPUT_FIRE, а каждый кадр, игрок выпускает пулю каждую 1/50 секунды, пока удерживает кнопку. Это может быть намеренным для скорострельного оружия, но для одиночного выстрела или выбора в меню нужно детектирование фронтов.

Техника: сохраняем ввод предыдущего кадра рядом с текущим и XOR-им их, чтобы найти изменившиеся биты:

```
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
```

Теперь input_pressed имеет единичный бит только для кнопок, которые *не были* нажаты в прошлом кадре, но *нажаты* в этом. Используй input_flags для непрерывных действий (движение) и input_pressed для одноразовых действий (выстрел, прыжок, выбор в меню).

Agon Light 2: PS/2-клавиатура через MOS

Agon читает свою PS/2-клавиатуру через MOS (Machine Operating System) API. eZ80 не сканирует клавиатурную матрицу напрямую — вместо этого сопроцессор ESP32 VDP обрабатывает аппаратное обеспечение клавиатуры и передаёт события нажатий на eZ80 через общий буфер.

Системная переменная MOS sysvar_keyascii (по адресу \$0800 + смещение) хранит ASCII-код последней нажатой клавиши или 0, если ни одна клавиша не нажата. Для игрового управления обычно опрашивают эту переменную или используют вызовы MOS API waitvblank / 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

```

Agon также поддерживает чтение состояний отдельных клавиш через VDU-команды (VDU 23,0,\$01,keycode), которые возвращают, удерживается ли конкретная клавиша в данный момент. Это ближе к подходу полуурядов Spectrum и лучше подходит для игр, которым нужно одновременное обнаружение нажатий. MOS API обрабатывает протокол PS/2, трансляцию сканкодов и автоповтор — обо всём этом тебе не нужно беспокоиться.

18.4 Структура сущности

Игровая сущность — это всё, что двигается, анимируется, взаимодействует или требует покадрового обновления: персонаж игрока, враги, пули, взрывы, всплывающие числа очков, усиления. На Z80 мы представляем каждую сущность как блок байтов фиксированного размера в памяти.

Раскладка структуры

Вот структура сущности, которую мы будем использовать во всех главах о разработке игр:

```

----- -----
+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

```

Битовые флаги в байте flags:

```

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

```

Почему 10 байт?

Десять байт — осознанный выбор. Это достаточно мало, чтобы 16 сущностей занимали всего 160 байт — ничтожно в терминах памяти. Важнее то, что умножение индекса сущности на 10 для нахождения смещения — простая задача на Z80:

```

; 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

```

Умножение на 10 использует стандартную декомпозицию: $10 = 8 + 2$. Мы вычисляем индекс $\times 2$, сохраняем его, вычисляем индекс $\times 8$ и складываем. Никакой реальной инструкции умножения — только сдвиги (ADD HL,HL) и сложение.

Если выбрать размер, равный степени двойки — 8 или 16 байт на сущность — вычисление индекса было бы ещё проще (три сдвига для 8, четыре для 16). Но 8 байт слишком тесно — ты потеряешь либо скорость, либо здоровье, и то и другое важно. А 16 байт тратят 6 байт на сущность на паддинг, что накапливается: 16 сущностей \times 6 потерянных байт = 96 байт мёртвого пространства. На Spectrum каждый байт на счету. Десять байт — правильный размер для данных, которые нам реально нужны.

Почему 16-битный X, но 8-битный Y?

Позиция X — 16-битная с фиксированной точкой (формат 8.8): старший байт — столбец пикселя (0–255), младший байт — субпиксельная дробная часть для плавного движения. Это необходимо для горизонтально скроллящихся игр, где персонаж движется с дробной скоростью. Персонаж, движущийся со скоростью 1.5 пикселя за кадр только с целочисленными координатами, чередовал бы шаги в 1 и 2 пикселя, создавая видимые рывки. С фиксированной точкой 8.8 движение плавное: добавляем 0x0180 к X каждый кадр, и позиция

пикселя продвигается 1, 2, 1, 2, 1, 2... по шаблону, который глаз воспринимает как стабильные 1.5 пикселя за кадр.

Позиция Y — всего 8 бит, потому что экран Spectrum имеет высоту 192 пикселя — одного байта достаточно для всего диапазона. Для игры с вертикальным скроллингом ты бы расширил Y до 16-битной фиксированной точки, ценой одного дополнительного байта на сущность.

Система фиксированной точки 8.8

Арифметика с фиксированной точкой была введена в Главе 4. Вот краткий обзор того, как она применяется к движению сущностей:

```
; 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
```

Красота фиксированной точки: сложение и вычитание — это обычные 16-битные операции ADD HL,DE. Никакой специальной обработки, никаких таблиц подстановки, никакого умножения. Дробная точность достигается автоматически, потому что мы несём субпиксельные биты с собой.

18.5 Массив сущностей

Сущности живут в статически выделенном массиве. Никакого динамического выделения памяти, никаких связных списков, никакой кучи. Статические массивы — стандартный подход на Z80 по уважительной причине: они быстрые, предсказуемые и не могут фрагментироваться.

```
; 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
```

Распределение слотов сущностей

Слот 0 — всегда игрок. Слоты 1–8 — враги. Слоты 9–15 — снаряды и эффекты (пули, взрывы, всплывающие очки). Это фиксированное разделение упрощает код: когда нужно перебрать врагов для ИИ, ты перебираешь слоты 1–8. Когда нужно создать пулью, ты ищешь в слотах 9–15. Игрок всегда по известному адресу.

```
; 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
```

Перебор сущностей

Основной цикл обновления проходит по каждому слоту сущности, проверяет флаг ACTIVE и вызывает соответствующий обработчик обновления:

```
; 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
```

Здесь IX используется как указатель на сущность, что удобно, потому что

IX-индексированная адресация позволяет обращаться к любому полю по смещению: $(IX+0)$ — X low, $(IX+2)$ — Y, $(IX+3)$ — type, и так далее. Минус IX — стоимость: каждая $LD A, (IX+n)$ стоит 19 тактов против 7 для $LD A, (HL)$. Для цикла обновления сущностей, который выполняется 16 раз за кадр, эти накладные расходы приемлемы. Для внутреннего цикла рендеринга, где данные сущности трогаются тысячи раз за кадр, ты бы сначала скопировал нужные поля в регистры.

Диспачт обновления по типу

У каждого типа сущности свой обработчик обновления. Мы используем ту же технику таблицы переходов, что и для конечного автомата игры:

```
; 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
```

Каждый обработчик получает IX, указывающий на сущность, и может обращаться ко всем полям через индексированную адресацию. Когда обработчик выполняет RET, он возвращается в цикл обновления сущностей, который переходит к следующему слоту.

Обработчик обновления игрока

Вот типичное обновление игрока — чтение флагов ввода, применение движения, обновление анимации:

```

; 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

```

Это намеренно просто. Глава 19 добавит гравитацию, прыжки и реакцию на столкновения. Пока что суть — в *структуре*: указатель сущности в IX, поля по смещению, флаги ввода управляют изменениями состояния, счётчик анимации тикает.

18.6 Пул объектов

Пули, взрывы и эффекты частиц — временные. Пуля существует долю секунды, прежде чем попасть во что-то или покинуть экран. Взрыв анимируется 8-16 кадров и исчезает. Можно создавать их динамически, но на Z80 “динамически” означает поиск свободной памяти, управление выделением и риск фрагментации. Вместо этого мы используем **пул объектов**: фиксированный набор слотов, в которые сущности активируются и из которых деактивируются.

У нас уже есть пул — это массив сущностей. Слоты 9–15 — пул снарядов/эффектов. Создание пули означает поиск неактивного слота в этом диапазоне и его заполнение. Уничтожение пули означает очистку её флага ACTIVE.

Создание пули

```

; 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

.found_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        ; 12T
    ld  (ix + 6), -4          ; dx = -4 (fast, leftward)
    jr  .set_flags             ; 12T

.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

```

Деактивация сущности

Когда пуля покидает экран или взрыв завершает анимацию, деактивация — одна инструкция:

```
; Deactivate entity at IX
deactivate_entity:
    ld    (ix + 9), 0      ; 19T  clear all flags (ACTIVE=0)
    ret
```

Вот и всё. В следующем кадре цикл обновления увидит ACTIVE=0 и пропустит слот. Слот теперь доступен для следующего вызова `spawn_bullet`.

Обработчик обновления пули

```
; 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
```

Размер пула

Семь слотов для снарядов (индексы 9–15) могут показаться ограничением. На практике этого более чем достаточно для большинства игр на Spectrum. Подумай: пуля, пересекающая весь экран (256 пикселей) со скоростью 4 пикселя

за кадр, летит 64 кадра — больше секунды. Если игрок стреляет раз в 8 кадров (высокая скорострельность), одновременно может существовать максимум 8 пуль. Семь слотов с редкими неудачами спавна (пуля просто не вылетает в этом кадре) ощущаются естественно, а не как баг. Игрок вряд ли заметит пропущенную пулю на пределе скорострельности.

Если нужно больше — расширяй массив сущностей. Но помни о стоимости: каждая дополнительная сущность добавляет ~160 тактов к худшему случаю цикла обновления (когда активна) и ~50 тактов даже когда неактивна (проверка флага ACTIVE и продвижение IX всё равно выполняются). Тридцать две сущности при всех активных потребляют примерно 16 000 тактов только в цикле обновления — четверть бюджета кадра до того, как ты отрисовал хоть один пиксель.

На Agon можно позволить себе большие пулы. При 360 000 тактах на кадр и аппаратном рендеринге спрайтов 64 или даже 128 сущностей вполне реальны.

18.7 Сущности взрывов и эффектов

Взрывы, всплывающие очки и эффекты частиц используют те же слоты сущностей, что и пули. Разница — в их обработчиках обновления: они проходят через последовательность кадров анимации и затем самоуничтожаются.

```
; 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
```

Чтобы создать взрыв при гибели врага:

```
; 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 + (SL0T_PR0J_FIRST * ENTITY_SIZE)
    ld    d, SL0T_PR0J_LAST - SL0T_PR0J_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

```

Паттерн всегда один и тот же: найти свободный слот, заполнить структуру, установить флаги. Обработчик обновления выполняет специфичную для типа работу. Деактивация очищает флаги. Слот переиспользуется в следующий раз, когда что-то нужно создать. Это весь жизненный цикл динамических объектов на Z80 — никакого аллокатора, никакого сборщика мусора, никакого списка свободных блоков. Просто массив и флаг.

18.8 Собираем всё вместе: каркас игры

Вот полный каркас игры, связывающий всё воедино. Это компилируемый фреймворк со всеми частями: конечный автомат, ввод, система сущностей и главный цикл.

```

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

```

Этот каркас компилируется, запускается и делает что-то видимое: цветные блоки перемещаются по сетке атрибутов. Блок игрока реагирует на управление QAO.P. Нажатие SPACE создаёт пули, которые летят через экран. Враги отскакивают между краями экрана. Когда пуля покидает экран, её слот освобождается для следующего выстрела.

Выглядит некрасиво — атрибутные блоки вместо спрайтов, нет скроллинга, нет звука. Но архитектура полна. Каждая часть из этой главы присутствует и подключена: главный цикл на HALT, диспетчер конечного автомата, считыватель ввода с детектированием фронтов, массив сущностей с обработчиками обновления по типам, и пул объектов для снарядов. Главы 16 и 17 предоставляют рендеринг. Глава 19 — физику и столкновения. Глава 11 — музыку. Этот каркас — место, куда они все подключаются.

18.9 Agon Light 2: та же архитектура, больше пространства

Agon Light 2 использует ту же фундаментальную структуру игрового цикла. eZ80 выполняет Z80-код нативно (в Z80-совместимом режиме или режиме ADL), так что главный цикл на HALT, конечный автомат, система сущностей и логика ввода — всё переносится напрямую.

Ключевые отличия:

Синхронизация кадров. Agon использует вызов MOS `waitvblank` (RST \$08, функция \$1E) вместо HALT для синхронизации кадров. VDP генерирует сигнал вертикального гашения, и MOS API его предоставляет.

Ввод. Чтение клавиатуры проходит через системные переменные MOS, а не через прямой ввод-вывод портов. Матрицы полурядов не существует — PS/2-клавиатура обрабатывается ESP32 VDP. Уровень абстракции ввода, который мы построили (всё сводится к `input_flags`), означает, что остальной код игры не заботится о разнице.

Бюджет сущностей. При ~360 000 тактах на кадр и аппаратном рендеринге спрайтов цикл обновления сущностей больше не является узким местом. Можно обновлять 64 сущности со сложным ИИ и всё равно использовать менее 10% бюджета кадра. Ограничивающий фактор на Agon — количество спрайтов

VDP на строку развёртки (обычно 16–32 аппаратных спрайта на одной линии), а не процессорное время.

Рендеринг. VDP Agon обрабатывает рендеринг спрайтов. Вместо ручного блиттинга пикселей в экранную память (шесть методов из Главы 16) ты отправляешь VDU-команды для позиционирования аппаратных спрайтов. Стоимость CPU на спрайт падает с ~1200 тактов (блит OR+AND на Spectrum) до ~50–100 тактов (отправка VDU-команды позиции). Это освобождает огромное количество процессорного времени для игровой логики.

Память. У Agon 512 КБ плоской памяти — никакого банкинга, никаких спорных регионов. Массив сущностей, таблицы подстановки, данные спрайтов, карты уровней и музыка могут сосуществовать без гимнастики переключения банков, описанной в Главе 15 для Spectrum 128K.

Практический вывод: на Agon архитектура этой главы масштабируется без усилий. Больше сущностей, более сложные конечные автоматы, больше логики ИИ — ничто из этого не угрожает бюджету кадра. Дисциплина подсчёта каждого такта по-прежнему важна (это хорошая инженерия), но ограничения, вынуждающие мучительные компромиссы на Spectrum, просто не применяются.

18.10 Проектные решения и компромиссы

Фиксированная vs переменная частота кадров

Главный цикл этой главы предполагает фиксированные 50 fps: сделай всё за один кадр или пропусти. Альтернатива — переменный временной шаг: измерить длительность кадра и масштабировать всё движение по дельте времени. Переменные временные шаги стандартны в современных игровых движках, но добавляют сложность на Z80 — нужен таймер кадра, умножение на дельту в каждом вычислении движения и аккуратная обработка стабильности физики при переменных скоростях.

Для игр на Spectrum фиксированные 50 fps — почти всегда правильный выбор. Аппаратура детерминирована, бюджет кадра предсказуем, и простота физики с фиксированным шагом (всё движется на постоянные величины каждый кадр) устраняет целую категорию багов. Если твоя игра падает ниже 50 fps, ответ — оптимизировать, пока не перестанет, а не добавлять переменный временной шаг.

На Agon, с его большим бюджетом, переменный тайминг ещё менее вероятно потребуется. Фиксируй частоту кадров на 50 или 60 fps и упрощай жизнь.

Размер сущности: компактный vs щедрый

Наша 10-байтная структура сущности — компактная. Некоторые коммерческие игры для Spectrum использовали 16 или даже 32 байта на сущность, храня дополнительные поля: предыдущая позиция (для стирания грязных прямоугольников), адрес спрайта, размеры коллизионного бокса, таймер ИИ и многое другое.

Компромисс — скорость перебора *versus* доступ к полям. Наш массив из 16 сущностей занимает 160 байт, и полный цикл обновления выполняется за ~8 000 тактов. Структура в 32 байта с 16 сущностями занимает 512 байт (всё ещё мало), но накладные расходы на перебор растут, потому что IX продвигается на 32 каждый шаг, и индексированные обращения к полям с большими смещениями вроде (*IX+28*) стоят те же 19 тактов, но их труднее отслеживать.

Если нужно больше данных на сущность, рассмотри разделение структуры: компактный “горячий” массив (позиция, тип, флаги — поля, используемые каждый кадр) и параллельный “холодный” массив (адрес спрайта, состояние ИИ, значение очков — поля, к которым обращаются только по необходимости). Это тот же компромисс “структура массивов” vs “массив структур”, с которым сталкиваются современные игровые движки, применённый в масштабе Z80.

Когда использовать HL вместо IX

IX-индексированная адресация удобна, но дорога: 19 тактов на обращение против 7 для (HL). В цикле обновления (вызывается 16 раз за кадр) накладные расходы IX приемлемы — 16 x 12 дополнительных тактов на обращение = 192 такта, ничтожно.

Но в цикле рендеринга, где ты можешь трогать 4–6 полей сущности для каждого из 8 видимых спрайтов, стоимость накапливается. Техника: в начале прохода рендеринга для каждой сущности копируй нужные поля в регистры:

```
; 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
```

Четыре обращения через IX по 19T = 76T авансом, затем вся процедура рендеринга использует обращения к регистрам по 4T вместо 19T через IX. Если процедура рендеринга трогает эти поля 10 раз, ты экономишь $(19-4) \times 10 - 76 = 74$ такта на сущность. Немного, но для 8 сущностей за кадр это 592 такта — достаточно, чтобы нарисовать ещё полспрайта.

Итого

- **Игровой цикл** — это HALT -> Ввод -> Обновление -> Рендеринг -> повтор. Инструкция HALT синхронизируется с прерыванием кадра, давая тебе ровно один кадр тактов (примерно 64 000 на Pentagon, 360 000 на Agon).
- **Конечный автомат** с таблицей переходов из указателей на обработчики (DW state_title, DW state_game, и т.д.) организует поток от титульного экрана через геймплей к экрану проигрыша. Диспачт стоит 73 такта независимо от количества состояний — постоянное время, чистое масштабирование.
- **Чтение ввода** на Spectrum использует IN A,(C) для опроса полурядов клавиатуры через порт \$FE. Пять клавиш (QAOP + SPACE) стоят примерно

220 тактов. Джойстик Kempston — одно чтение порта за 11Т. Детектирование фронтов (нажатие vs удержание) использует XOR между текущим и предыдущим кадрами.

- **Структура сущности** — 10 байт: X (16-бит с фиксированной точкой), Y, тип, состояние, anim_frame, dx, dy, здоровье, флаги. Шестнадцать сущностей занимают 160 байт. Умножение на 10 для преобразования индекса в адрес использует декомпозицию $10 = 8 + 2$.
 - **Массив сущностей** статически выделен с фиксированным распределением слотов: слот 0 для игрока, слоты 1–8 для врагов, слоты 9–15 для снарядов и эффектов. Перебор проверяет флаг ACTIVE и диспатчит к обработчикам по типам через вторую таблицу переходов.
 - **Пул объектов** — это сам массив сущностей. Создание устанавливает поля и флаг ACTIVE. Деактивация очищает флаги. Поиск свободного слота — линейный сканирование соответствующего диапазона слотов. Семь слотов для снарядов справляются с типичной скорострельностью без того, чтобы игрок замечал пропущенные спавны.
 - **IX-индексированная адресация** удобна для доступа к полям сущности (19Т на обращение), но дорога во внутренних циклах. Копирай поля в регистры в начале рендеринга для обращений по 4Т.
 - Agon Light 2 использует ту же архитектуру с большим запасом. MOS waitvblank заменяет HALT, PS/2-клавиатура заменяет сканирование полу-рядов, аппаратные спрайты заменяют CPU-блиттинг. Цикл обновления сущностей больше не является узким местом.
 - Практический каркас в этой главе содержит конечный автомат, 16 сущностей (1 игрок + 8 врагов + 7 слотов для пуль/эффектов), ввод с детектированием фронтов, обработчики обновления по типам и минимальный рендерер атрибутных блоков. Это шасси, к которому подключаются Главы 16 (спрайты), 17 (скроллинг), 19 (столкновения) и 11 (звук).
-

Попробуй сам

1. **Собери каркас.** Скомпилируй каркас игры из раздела 18.8 и запусти его в эмуляторе. Используй QAOB для перемещения блока игрока и SPACE для стрельбы. Наблюдай, как цветные блоки двигаются. Добавь профилирование цветом бордюра (Глава 1), чтобы увидеть, сколько бюджета кадра используется.
2. **Добавь шестое состояние.** Реализуй STATE_SHOP между Menu и Game. Экран магазина должен отображать три предмета и позволять игроку выбрать один клавишами UP/DOWN и FIRE. Это упражнение на конечный автомат — добавь константу, запись в таблице, обработчик и логику перехода.
3. **Увеличь количество сущностей.** Увеличь MAX_ENTITIES до 32, добавь 16 врагов и измерь влияние на бюджет кадра с помощью профилирования бордюром. При каком количестве сущностей цикл обновления начинает угрожать 50 fps?

4. **Реализуй поддержку Kempston.** Добавь считыватель джойстика Kempston и объедини его с клавиатурным вводом через OR. Протестируй в эмуляторе, поддерживающем эмуляцию Kempston (Fuse: Options -> Joysticks -> Kempston).
5. **Раздели горячие и холодные данные.** Создай второй “холодный” массив с 4 байтами на сущность (адрес спрайта, таймер ИИ, состояние ИИ, значение очков). Измени цикл обновления так, чтобы холодные данные использовались только когда типа сущности этого требует (враги для ИИ, не пули). Измерь экономию тактов.

Далее: Глава 19 — Столкновения, физика и ИИ врагов. Мы добавим AAVB-обнаружение столкновений, гравитацию, прыжки и четыре модели поведения врагов к каркасу из этой главы.

Источники: клавиатурная матрица и раскладка портов ZX Spectrum из технической документации Sinclair Research; спецификация интерфейса джойстика Kempston; техники арифметики с фиксированной точкой из Главы 4 (Dark/X-Trade, Spectrum Expert #01, 1997); профилирование цветом бордюра из Главы 1; методы рендеринга спрайтов из Главы 16; интеграция музыки AY из Главы 11; документация MOS API Agon Light 2 (Bernardo Kastrup, agon-light.com)

Глава 19: Столкновения, физика и ИИ врагов

“Каждая игра — это ложь. Физика подделана. Интеллект — это поиск по таблице. Игрок не замечает, потому что ложь рассказывается со скоростью 50 кадров в секунду.”

В Главе 18 мы построили игровой цикл, систему сущностей, отслеживающую шестнадцать объектов, и обработчик ввода. Но прямо сейчас наш игрок проходит сквозь стены, парит над землёй, а враги стоят на месте. Игра без столкновений — это заставка. Игра без физики — это головоломка со сдвигом. Игра без ИИ — это песочница, в которой нечему сопротивляться.

Эта глава добавляет три системы, превращающие техническое демо в игру: обнаружение столкновений, физику и ИИ врагов. Все три разделяют философию проектирования: подделай достаточно хорошо, достаточно быстро, и никто не заметит разницы. Мы опираемся на структуру сущности из Главы 18 — 16-байтную запись с позициями X/Y в формате 8.8 с фиксированной точкой, скоростью в dx/dy, типом, состоянием и флагами.

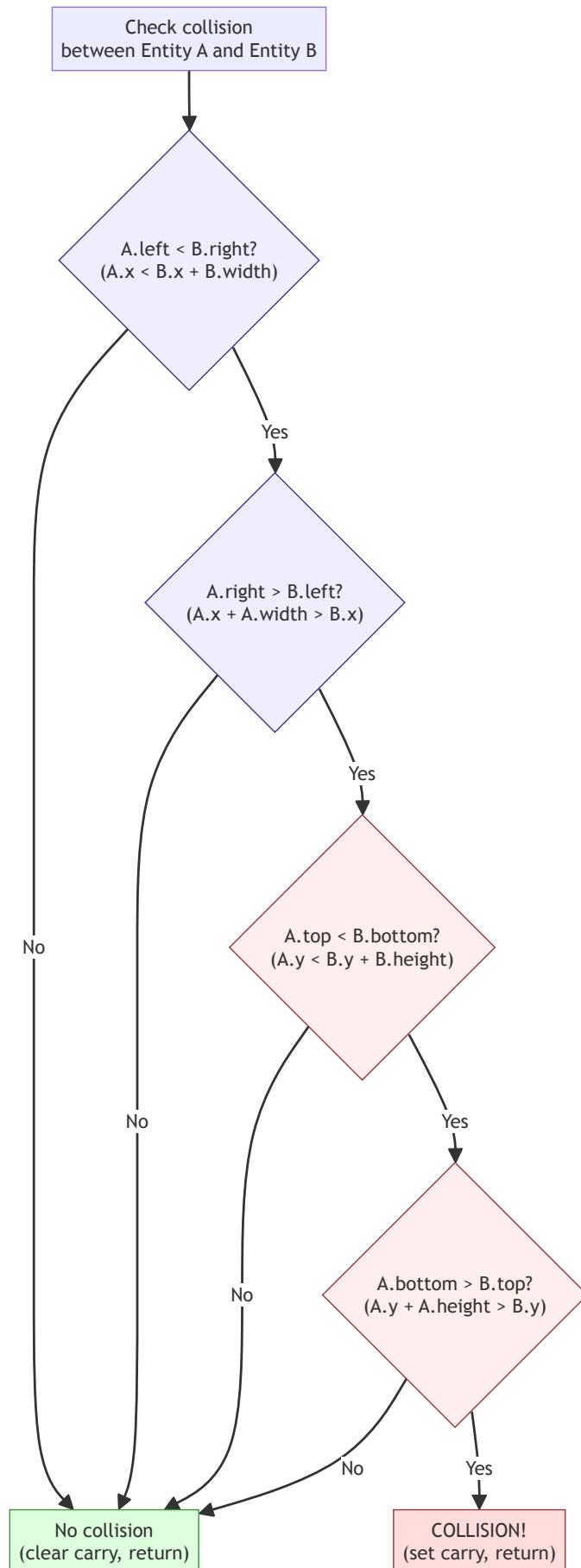
Часть 1: Обнаружение столкновений

ААВВ: единственная форма, которая тебе нужна

Ограничивающие прямоугольники, выровненные по осям. Каждая сущность получает прямоугольник, определённый её позицией и размерами: левый край, правый край, верхний край, нижний край. Два прямоугольника перекрываются тогда и только тогда, когда все четыре условия истинны:

1. Левый край А меньше правого края В
2. Правый край А больше левого края В
3. Верхний край А меньше нижнего края В
4. Нижний край А больше верхнего края В

Если хоть одно из этих условий не выполняется, прямоугольники не перекрываются. Это **ранний выход**, который делает ААВВ быстрым: в среднем большинство пар сущностей не сталкиваются, поэтому большинство проверок завершаются после одного-двух сравнений, а не выполняют все четыре.



Ранний выход экономит такты: Большинство пар сущностей находятся далеко друг от друга. Первый тест на пересечение по X отклоняет их за ~91 такт (T-state). Только пары, прошедшие все четыре теста (худший случай: ~270 тактов), являются реальными столкновениями. В сайд-скроллерах проверяй горизонтальное пересечение первым — сущности более разбросаны по X, чем по Y.

На Z80 мы храним позиции сущностей как значения 8.8 с фиксированной точкой, но для обнаружения столкновений нам нужна только целая часть — старший байт каждой координаты. Пописельной точности более чем достаточно. Вот полная процедура AABB-столкновения:

```
; 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

```



Рис. 39: Тест AABB-обнаружения столкновений с двумя сущностями, цвет бордюра меняется при пересечении

IX/IY-индексированная адресация удобна, но дорога — 19 тактов на обращение против 7 для `ld a, (hl)`. Для игры с 8 врагами и 7 пулями это приемлемо. Худ-

ший случай (все четыре теста пройдены, столкновение обнаружено): примерно 270 тактов. Лучший случай (первый тест провален): примерно 91 такт. Для 8 врагов, проверяемых против игрока, средний случай — примерно $8 \times 120 = 960$ тактов — 1.3% бюджета кадра Pentagon. Столкновения дёшевы.

Внимание, переполнение: Инструкции ADD A, (ix+13) вычисляют $x + width$ в 8-битном регистре. Если сущность расположена на $X=240$ с шириной 24, результат переполняется до 8, что приводит к некорректным сравнениям. Убедись, что позиции сущностей ограничены так, чтобы $x + width$ и $y + height$ никогда не превышали 255 — обычно достаточно ограничить игровую область, оставив запас у правого и нижнего краёв. Как вариант, можно перейти к 16-битной арифметике сравнения за счёт дополнительных инструкций.

Порядок тестов для быстрейшего отклонения

Порядок имеет значение. В боковом скроллере сущности, далёкие друг от друга по горизонтали — типичный случай. Тестирование горизонтального перекрытия первым отклоняет их после двух сравнений. Можно пойти дальше с быстрым предварительным отклонением:

```
; 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:
```

Это предварительное отклонение стоит около 60 тактов, экономя 82+ такта полной AABB-проверки. На скроллящемся уровне обычно лишь 2-3 врага достаточно близки, чтобы потребовать полного теста.

Столкновения с тайлами: тайлмэп как поверхность столкновений

В платформере игрок сталкивается с миром — полами, стенами, потолками, шипами. Мы используем сам тайлмэп как таблицу подстановки: преобразуем пиксельную позицию в индекс тайла, ищем тип тайла, ветвимся по результату. Один поиск в массиве заменяет десятки проверок прямоугольников.

Предположим тайлмэп 32x24 с тайлами 8x8 пикселей (естественная знакосетка Spectrum):

```
; 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
```

Теперь проверяем углы и рёбра сущности против тайлмэпа:

```
; 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

```

Ключевое понимание: поиск точки в тайле — это $O(1)$ обращение к массиву. Каждый вызов `tile_at` стоит ~182 такта (T-state). Вся система столкновений с тайлами (проверка ног, головы, левого и правого краёв) стоит примерно 800–1 200 тактов на сущность, независимо от размера карты.

Скользящая реакция на столкновение

Когда игрок врезается в стену, двигаясь по диагонали, он должен *скользить*, а не останавливаться. Разрешай столкновения по каждой оси независимо:

1. Применить горизонтальную скорость. Проверить горизонтальные столкновения с тайлами. Если заблокировано — вытолкнуть и обнулить горизонтальную скорость.
2. Применить вертикальную скорость. Проверить вертикальные столкновения с тайлами. Если заблокировано — вытолкнуть и обнулить вертикальную скорость.

Именно это и делает `check_player_tiles` — каждая ось обрабатывается отдельно. Диагональное движение вдоль стены естественно превращается в скольжение. Большинство платформеров применяют сначала X (управляемый игроком), затем Y (гравитация). Поэкспериментируй с обоими порядками и почувствуй разницу.

Часть 2: Физика

То, что мы строим — не симуляция твёрдого тела, а небольшой набор правил, создающих *ощущение* веса и инерции. Три операции покрывают 90% того, что нужно платформеру: гравитация, прыжок и трение.

Гравитация: убедительное падение

Каждый кадр добавляем константу к вертикальной скорости сущности:

```
; 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
```

Представление с фиксированной точкой из Главы 4 делает здесь тяжёлую работу. Гравитация — 0.25 пикселя на кадр в квадрате, значение, которое было бы невозможно представить целочисленной арифметикой. В формате 8.8 это просто \$0040. Каждый кадр dy растёт на 0.25. Через 4 кадра сущность падает со скоростью 1 пиксель за кадр. Через 16 кадров — 4 пикселя за кадр (предельная скорость). Кривая ускорения выглядит естественно, потому что она **является** естественной — постоянное ускорение — это просто линейное приращение скорости.

Ограничение предельной скорости предотвращает падение сущностей настолько быстро, что они проскаивают сквозь полы (проблема “туннелирования”). Максимальная скорость падения 4 пикселя за кадр означает, что сущность никогда не сдвинется больше чем на половину высоты тайла за один кадр, так что проверки столкновений с тайлами всегда её поймают.

Почему фиксированная точка важна здесь

Без фиксированной точки гравитация — это либо 0, либо 1 пиксель за кадр: пух или камень, ничего между ними. Фиксированная точка 8.8 даёт 256 значений между каждым целым числом. \$0040 (0.25) создаёт плавную дугу. \$0080 (0.5) ощущается тяжело. \$0020 (0.125) ощущается как прыжок на Луне. Настройка этих констант — это то, где твоя игра обретает характер. Если основы фиксированной точки расплывчаты, перечитай Главу 4.

Прыжок: антигравитационный импульс

Прыжок — простейшая операция физики в игре: установить вертикальную скорость в большое отрицательное значение (вверх). Гравитация замедлит её, доведёт до нуля в верхней точке и потянет обратно вниз. Дуга прыжка — естественная парабола, никакого явного расчёта дуги не нужно.

```
; 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

```

При гравитации 0.25/кадр² и силе прыжка -3.5/кадр игрок поднимается 14 кадров до пика примерно в 24 пикселя (~3 тайла), затем падает ещё 14 кадров. Общее время в воздухе: 28 кадров, чуть больше полсекунды. Отзывчиво, но не дёргано.

Прыжки переменной высоты

Если игрок отпускает кнопку прыжка во время подъёма, урежь скорость подъёма вдвое. Короткое нажатие даёт короткий прыжок, удержание — полный.

```

; 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

```

Это 16-битный арифметический сдвиг вправо: SRA сохраняет знак в старшем байте, RRA подхватывает перенос в младшем байте. Скорость подъёма уменьшается вдвое, дуга выравнивается. Сорок тактов за значительно лучше ощущающийся прыжок.

Трение: замедление на земле

Когда игрок отпускает клавиши направления, он должен замедляться, а не останавливаться мгновенно. Операция — один сдвиг вправо горизонтальной скорости.

```
; 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
```

Сдвиг вправо на 1 делит скорость на 2 каждый кадр — игрок останавливается за несколько кадров. Для льда применять трение реже:

```
; 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
```

Варьируй трение по типу поверхности — ищи тайл под ногами сущности и ветвись:

```
; 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:

```

Применение скорости к позиции

Заключительный шаг: перемещаем сущность на её скорость через 16-битное сложение с фиксированной точкой по каждой оси:

```

; 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

```

Цикл физики

Собирая всё вместе, покадровое обновление физики для одной сущности выглядит так:

```

; 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

```

Порядок намеренный: сначала силы, затем движение, затем столкновения. Это стандарт для платформеров. Общая стоимость на сущность: примерно 1 000–1 500 тактов (основная часть приходится на проверки столкновений с тайлами по ~182 такта каждая). Для 16 сущностей: 16 000–24 000 тактов, около 25–33% бюджета кадра Pentagon. На практике полная проверка столкновений с тайлами нужна только игроку и врагам, подверженным гравитации — пули и эффекты могут обойтись более простой проверкой границ.

Часть 3: ИИ врагов

У тебя нет тактов для поиска пути или деревьев решений. Но есть таблица переходов и байт состояния. Этого достаточно.

Конечный автомат

Каждый враг имеет байт `state` (смещение +5 в нашей структуре сущности), который выбирает, какая процедура поведения выполняется в этом кадре:

Состояние	Имя	Поведение
0	PATROL	Ходит туда-сюда между двумя точками
1	CHASE	Двигается к игроку
2	ATTACK	Стреляет снарядом или атакует
3	RETREAT	Двигается от игрока
4	DEATH	Проигрывает анимацию смерти, затем деактивируется

Переходы — простые условия: проверки близости, таймеры перезарядки, пороги здоровья. Каждый — сравнение или проверка бита, никогда ничего дорогостоящего.

Таблица JP

Ядро диспетчера ИИ — **таблица переходов**, индексируемая байтом состояния. Диспатч за O(1) независимо от количества состояний:

```
; 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
```

```

STCHASE equ 1
STATTACK equ 2
STRETREAT equ 3
STDEATH 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

```

Инструкция `jp (hl)` стоит всего 4 такта — общие накладные расходы на диспачт около 45 тактов, независимо от количества состояний. Примечание: `jp (hl)` переходит по адресу, находящемуся в `HL`, а не по адресу, на который указывает `HL`. Скобки — особенность нотации Zilog.

Patrol: тупой обход

Простейшее поведение ИИ: идти в одном направлении, пока не достигнешь границы, затем развернуться.

```

; 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), STCHASE ; 19T set state = CHASE
    ret                 ; 10T

```

Патрулирующий враг стоит около 120 тактов за кадр. Это ничтожно. Восемь патрулирующих врагов стоят менее 1 000 тактов — едва заметная точка в бюджете кадра.

Chase: неотступный преследователь

Поведение преследования просто: вычислить знак горизонтального расстояния между врагом и игроком и двигаться в этом направлении.

```
; 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
```

Техника определения знака dx: вычитание, проверка переноса. Перенос установлен — значит игрок слева. Две инструкции, никакой тригонометрии, никакого поиска пути.

Attack: выстрел и перезарядка

Состояние ATTACK выпускает снаряд, затем ждёт таймер перезарядки. Мы переиспользуем поле anim_frame (смещение +6) как обратный отсчёт.

```
; 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), STCHASE        ; back to chase state
    ret

```

Процедура `find_free_entity` (из Главы 18) сканирует неактивный слот. Если пул полон, выстрел пропускается.

Retreat: обратное преследование

Зеркало `chase` — вычислить знак `dx`, двигаться в другую сторону:

```

; 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: анимация и удаление

Здоровье достигает нуля, состояние становится DEATH. Обработчик проигрывает анимацию, затем деактивирует сущность.

```

; 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

```

Как только бит 7 очищен, сущность исчезает из рендеринга и её слот становится доступным для переиспользования.

Оптимизация: обновление ИИ каждый 2-й или 3-й кадр

Игроки не отличают ИИ на 50 Гц от ИИ на 25 Гц. Экран и ввод игрока работают на 50 fps, но решения врагов на 25 fps (каждый 2-й кадр) или 16.7 fps (каждый 3-й) неотличимы. Скорость переносит сущность плавно между тиками ИИ.

```

; 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

```

Это уменьшает стоимость ИИ вдвое. Для обновления каждый 3-й кадр используй проверку модуля 3:

```

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

```

Ключевое понимание: физика работает каждый кадр для плавного движения. ИИ работает каждый 2-й или 3-й кадр для решений. Игрок видит плавное движение с чуть задержанными реакциями, и результат ощущается естественно.

Часть 4: Практика — четыре типа врагов

Четыре типа врагов, каждый с отдельным поведением, подключенные к системе сущностей из Главы 18.

1. Walker — патрулирует платформу, разворачивается у краёв. Обнаруживает игрока по близости. Поведение преследования: следовать на уровне земли. Урон: только контактный (без снарядов). Здоровье: 1 попадание.

Состояние	Поведение	Переход
PATROL	Ходит влево/вправо в пределах диапазона	Игрок в 48 пикселях: CHASE
CHASE	Двигается к игроку на 2x скорости	В 16 пикселях: ATTACK
ATTACK	Пауза, бросок вперёд	Кулдаун истёк: CHASE
DEATH	Мигание 8 кадров, деактивация	-

2. Shooter — стоит на месте (или медленно патрулирует), стреляет снарядами, когда игрок в зоне досягаемости. Держит дистанцию.

Состояние	Поведение	Переход
PATROL	Медленное движение или неподвижность	Игрок в 64 пикселях: ATTACK
ATTACK	Выстрел пулей, кулдаун 30 кадров	Игрок вне зоны: PATROL
RETREAT	Отступление, если игрок слишком близко	Дистанция > 32 пикс.: ATTACK
DEATH	Анимация взрыва, деактивация	-

3. Swooper — движется вертикально по синусоиде (или простой вверх/вниз), пикирует к игроку при совмещении.

```
; 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), STCHASE
ret

```

Swooper использует таблицу синусов из Главы 4 для вертикальных колебаний. Когда игрок проходит снизу, враг пикирует.

4. Ambusher — сидит в засаде, пока игрок не окажется очень близко, затем агрессивно активируется.

```

; 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), STCHASE ; go straight to aggressive chase
    ; (Could also play an activation sound/animation here)
    ret

```

Манхэттенское расстояние ($|dx| + |dy|$) стоит около 30 тактов против ~ 200 для евклидова. Для проверок близости этого достаточно.

Подключение к игровому циклу

Полное покадровое обновление, построенное на Главе 18:

```

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

```

Процедура `check_all_collisions` проверяет столкновения игрока с врагами и пуль с существами:

```

; 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

```

Заметки по Agon Light 2

Тот же код физики и ИИ работает на Agon без изменений — чистая Z80-арифметика без аппаратных зависимостей. Бюджет Agon в ~368 000 тактов означает, что ты можешь позволить себе больше сущностей (32 или 64), ИИ каждый кадр (без пропуска каждого 2-го кадра), больше точек проверки столкновений и более богатые конечные автоматы. Сохраняй физические константы идентичными между платформами, чтобы игра *ощущалась* одинаково. VDP Agon предоставляет аппаратное обнаружение столкновений спрайтов для проверок пуля-против-врага, но столкновения с тайлами остаются поиском по тайлмэпу на Z80.

Руководство по настройке

Числа в этой главе — отправные точки, а не заповеди. Вот справочная таблица для настройки ощущений твоего платформера:

Параметр	Значение	Эффект
GRAVITY_FRAC	\$20 (0.125)	Воздушный, лунный
GRAVITY_FRAC	\$40 (0.25)	Стандартное ощущение платформера
GRAVITY_FRAC	\$60 (0.375)	Тяжёлый, быстро падающий
JUMP_INT	\$FD (-3)	Низкий прыжок (~2 тайла)
JUMP_INT:FRAC	\$FC:\$80 (-3.5)	Средний прыжок (~3 тайла)
JUMP_INT	\$FB (-5)	Высокий прыжок (~5 тайлов)
PATROL_SPEED	1	Медленный, предсказуемый

Параметр	Значение	Эффект
CHASE_SPEED	2	Совпадает со скоростью ходьбы игрока
CHASE_SPEED	3	Быстрее игрока — вынуждает прыгать
DETECT_RANGE	32	Короткая дальность, враг “тупой”
DETECT_RANGE	64	Средняя дальность, сбалансировано
DETECT_RANGE	128	Большая дальность, агрессивный враг
ATTACK_COOLDOWN	15	Быстрая стрельба (2 выстрела/сек при 25 Гц ИИ)
ATTACK_COOLDOWN	30	Умеренная скорострельность
ATTACK_COOLDOWN	60	Медленная, выверенная
Сдвиг трения	»1 каждый кадр	Остановка за ~3 кадра (липкое)
Сдвиг трения	»1 каждые 2 кадра	Остановка за ~6 кадров (гладкое)
Сдвиг трения	»1 каждые 4 кадра	Остановка за ~12 кадров (лёд)

Тестируй постоянно. Меняй одно число, играй тридцать секунд, почувствуй разницу. Настройка физики — это не инженерия, а ремесло. Числа должны быть в блоке констант в начале исходного файла, ясно подписаны, легко изменяемы.

Итого

- **АABB-столкновение** использует четыре сравнения с ранним выходом. Большинство пар отклоняются после одного-двух тестов. Стоимость: 91–270 тактов на пару на Z80 (доминирует IX/IY-индексированная адресация). Располагай тесты так, чтобы отклонять наиболее частый случай отсутствия столкновения первым (обычно горизонтальный). Следи за 8-битным переполнением при вычислении $x + width$ вблизи краёв экрана.
- **Столкновение с тайлами** преобразует пиксельные координаты в индекс тайла через сдвиг вправо и поиск. $O(1)$ на проверяемую точку, независимо от размера карты. Проверяй четыре угла и серединные точки рёбер ограничивающего прямоугольника сущности.
- **Скользящая реакция на столкновение** разрешает столкновения по каждой оси независимо. Примени горизонтальную скорость, затем проверь горизонтальные столкновения; примени вертикальную скорость, затем проверь вертикальные. Диагональное движение вдоль стены естественно становится скольжением.

- **Гравитация** — это сложение с фиксированной точкой к вертикальной скорости каждый кадр: $dy += gravity$. В формате 8.8 субпиксельные значения вроде 0.25 пикселя/кадр² создают плавные, естественные кривые ускорения.
 - **Прыжок** устанавливает вертикальную скорость в отрицательное значение. Гравитация замедляет её, создавая параболическую дугу без явного расчёта кривой. Прыжки переменной высоты урезают скорость вдвое при отпускании кнопки.
 - **Трение** — это сдвиг вправо горизонтальной скорости: $dx >= 1$. Варьируй частоту применения для разных типов поверхностей (каждый кадр = грувая земля, каждый 4-й кадр = лёд).
 - **ИИ врагов** использует конечный автомат с диспачем через JP-таблицу. Пять состояний (Patrol, Chase, Attack, Retreat, Death) покрывают большинство поведений врагов в платформере. Стоимость диспача: ~45 тактов, независимо от количества состояний.
 - **Chase** использует знак `player.x - enemy.x` для определения направления. Две инструкции, ноль тригонометрии.
 - **Обновляй ИИ каждый 2-й или 3-й кадр**, чтобы уменьшить стоимость CPU вдвое или втрое. Физика работает каждый кадр для плавного движения; решения ИИ могут отставать на 1-2 кадра без того, чтобы игрок это заметил.
 - **Четыре типа врагов** (Walker, Shooter, Swooper, Ambusher) демонстрируют, как один и тот же каркас конечного автомата создаёт разнообразное поведение через изменение нескольких констант и одного-двух обработчиков состояний.
 - **Общая стоимость** для игры с 16 сущностями (физика + столкновения + ИИ): примерно 15 000–20 000 тактов на кадр на Spectrum (около 25–28% бюджета кадра Pentagon), оставляя достаточно места для рендеринга и звука.
-

Попробуй сам

1. **Собери AABB-тест.** Размести две сущности на экране. Двигай одну клавиатурой. Меняй цвет бордюра при столкновении. Проверь поведение раннего выхода, разместив сущности далеко друг от друга и измерив такты тестовой связкой цвета бордюра из Главы 1.
2. **Реализуй столкновения с тайлами.** Создай простой тайлмэп с твёрдыми блоками и пустым пространством. Двигай игрока клавиатурным вводом и гравитацией. Проверь, что игрок приземляется на платформы, не может проходить сквозь стены и скользит вдоль поверхностей при диагональном движении.
3. **Настрой физику.** Используя таблицу настройки выше, отрегулируй гравитацию и силу прыжка для создания трёх разных ощущений: воздушный (Луна), стандартный (Mario-подобный) и тяжёлый (Castlevania-подобный). Поиграй в каждый минуту и отметь, как константы меняют ощущения.
4. **Построй все четыре типа врагов.** Начни с Walker (patrol + chase), затем добавь Shooter (снаряды), Swooper (движение по синусоиде) и Ambusher

(активация из засады). Протестируй каждого отдельно, прежде чем комбинировать их на одном уровне.

5. **Профилируй бюджет кадра.** Со всеми 16 активными сущностями используй многоцветный профилировщик бордюром (Глава 1), чтобы визуализировать, сколько кадра тратится на физику (красный), ИИ (синий), столкновения (зелёный) и рендеринг (жёлтый). Измени частоту обновления ИИ и измерь разницу.

Источники: Dark “Программирование алгоритмов” (Spectrum Expert #01, 1997) — основы арифметики с фиксированной точкой; фольклор разработки игр и платформенные знания Z80; техника таблицы переходов — стандартная практика Z80, документированная в сообществе разработчиков ZX Spectrum

Глава 20: Рабочий процесс создания демо — от идеи до компо

“Дизайн — это совокупность всех компонентов демо, как видимых, так и скрытых. Дизайн характеризует реализационную, стилистическую, идеиную целостность.” – Introspec, “О дизайне”, Нуре, 2015

Демо не создаётся за один сеанс вдохновенного кодинга. Это проект — с дедлайнами, зависимостями, творческими решениями, которые нужно зафиксировать за недели до пати, техническими ставками, которые оправдают себя или нет, и финальной работой, которая либо работает на машине для компо, либо ломается на глазах у зрителей. Расстояние от “у меня есть идея для демо” до “оно заняло третье место на DiHalt” измеряется не строками кода, а рабочим процессом: как ты организуешь эффекты, как планируешь время, как собираешь и тестируешь, как справляешься с неизбежным моментом, когда музыка не готова, а до пати четыре дня.

Эта глава — о том самом рабочем процессе. Мы потратили девятнадцать глав на техники — внутренние циклы, подсчёт тактов, сжатие, звук, синхронизацию. Теперь мы делаем шаг назад и спрашиваем: как реальное демо собирается воедино? Как перейти от пустого экрана к двухминутной продукции, которая надёжно работает, выглядит целенаправленно и попадает на нужную пати в нужный день?

Ответы приходят из трёх источников. Статья restorer'a *making-of* для Lo-Fi Motion (Нуре, 2020) — подробное исследование работающего производственного конвейера: четырнадцать эффектов, построенных за две недели вечернего кодинга, с системой таблицы сцен и тулчейном, который может воспроизвести любой читатель. Философские эссе Introspec'а на Нуре — “О дизайне” и “MORE” (оба 2015) — формулируют дизайнерское мышление, которое отделяет коллекцию эффектов от целостного демо. И более широкая культура *making-of* в ZX Spectrum-сцене — от подробного NFO Eager до рабочего процесса GABBA с видеоредактором на iOS и до 256-байтной головоломки NHBF — даёт нам галерею подходов для изучения.

20.1 Что означает “дизайн” в демо

Когда демосценеры говорят “дизайн”, они не имеют в виду графический дизайн. Они не имеют в виду компоновку UI или теорию цвета, хотя это тоже важно. Определение Introspec’а, опубликованное на Нуре в январе 2015, шире и требовательнее:

Дизайн — это совокупность всех компонентов демо, как видимых, так и скрытых.

Это определение включает эффекты, которые видит зритель, переходы между ними, выбор музыки и её синхронизацию с визуалом, цветовую палитру, ритм, эмоциональную дугу — но также архитектуру кода, раскладку памяти, стратегию сжатия, конвейер сборки и решения о том, что исключить. Демо с красивым пиксель-артом и ужасным ритмом имеет плохой дизайн. Демо с грубым визуалом, но идеальной музыкальной синхронизацией и ясной эмоциональной дугой может иметь превосходный дизайн. Намеренно уродливое демо, выбравшее свою эстетику осознанно, может иметь выдающийся дизайн.

Следствия для рабочего процесса немедленны. Если дизайн охватывает всё, то дизайнерские решения принимаются на каждом этапе. Выбор ассемблера ограничивает возможности конвейера сборки. Карта памяти определяет, какие эффекты могут существовать. Порядок, в котором ты строишь эффекты, определяет, что можно вырезать, если время кончается. Каждый технический выбор имеет эстетическое последствие, и каждый эстетический выбор имеет техническую стоимость.

Производство демо должно быть одновременно снизу вверх и сверху вниз, с постоянной обратной связью между творческим видением и технической реальностью. Рабочий процесс должен поддерживать этот цикл обратной связи.

20.2 Lo-Fi Motion: полное исследование случая

В сентябре 2020 restorer опубликовал статью making-of для Lo-Fi Motion, демо для ZX Spectrum, выпущенного на DiHalt 2020. Статья ценна не каким-то одним техническим открытием, а тем, что документирует *весь производственный конвейер* — от начальной концепции до готового бинарника — с достаточной детализацией для воспроизведения.

Концепция: “белорусский пиксель”

Lo-Fi Motion использует графику в разрешении атрибутов — то, что restorer называет “lo-fi”-рендерингом. Большинство эффектов работают на сетке атрибутов 32x24 или в удвоенном разрешении 32x48, используя полузнакоряды. Никакого попиксельного рендера. Эстетика намеренно блочная, принимающая атрибутную сетку ZX Spectrum, а не борющаяся с ней. Название говорит само за себя: это lo-fi, и движение — суть.

Это дизайнерское решение с каскадными техническими преимуществами. Эффекты в разрешении атрибутов дёшевы в расчёте (192 или 384 байта на кадр вместо 6 144), дёшевы в хранении (малые буферы кадра означают больше

места для сжатых данных) и быстры для отображения (запись 768 байт в память атрибутов легко умещается в один кадр). Lo-fi-эстетика — не компромисс, а выбор, который позволяет реализовать четырнадцать эффектов за две недели вечерней работы.

Таблица сцен

В центре архитектуры Lo-Fi Motion — **таблица сцен**, структура данных, управляющая всем демо. Каждая запись в таблице описывает одну сцену:

```
; 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
```

Движок демо читает таблицу сцен последовательно. Для каждой записи он подключает указанный банк памяти, переходит по адресу входа и запускает эффект на указанное количество кадров. Когда длительность истекает, он переходит к следующей записи. Всё демо — все четырнадцать эффектов, все переходы, весь тайминг — закодировано в одной таблице.

Это тот же архитектурный паттерн, который мы видели в скриптовом движке из Главы 12, сведённый к сути. Скриптовый движок Eager имел два уровня (внешний скрипт для эффектов, внутренний скрипт для вариаций параметров) и команду kWORK для асинхронной генерации кадров. Таблица сцен Lo-Fi Motion проще: один уровень, синхронная генерация, без асинхронной буферизации. Простота — суть. Это работает. Это было построено за две недели.

Паттерн таблицы сцен имеет критическое преимущество для рабочего процесса: он отделяет контент от движка. Добавление нового эффекта означает написание процедуры эффекта и добавление одной записи в таблицу. Перестановка порядка демо означает перестановку записей таблицы. Корректировка тайминга означает изменение значений длительности. Код движка не меняется. Это разделение означает, что можно итерировать структуру демо — его ритм, порядок, тайминг — не трогая движок, и итерировать отдельные эффекты, не трогая структуру.

Четырнадцать эффектов

Lo-Fi Motion содержит приблизительно четырнадцать различных визуальных эффектов. restorer перечисляет их по рабочим названиям: raskolbas, slime, fire, interp, plasma, rain, dina, rtzoomer, rbars, bigpic и несколько других. Каждый эффект — самодостаточная процедура, рендерящая в виртуальный буфер.

Виртуальный буфер — ключевое архитектурное решение. Большинство эффектов не пишут напрямую в экранную память. Вместо этого они рендерят в **буфер с одним байтом на пиксель** — блок RAM, где каждый байт представляет значение цвета одной ячейки атрибутов. Буфер обычно имеет ширину 32 байта и высоту 24 или 48 байт (для полузнакового разрешения). После того

как эффект отрисует в буфер, отдельная процедура вывода копирует буфер в память атрибутов, выполняя необходимое преобразование формата.

Эта косвенность стоит несколько сотен тактов за кадр, но даёт два преимущества. Во-первых, эффекты изолированы от физической раскладки экрана. Эффект, рендерящий в линейный буфер, не обязан знать о структуре адресации памяти атрибутов. Во-вторых, эффекты могут комбинироваться: два эффекта могут рендерить в отдельные буфера, а процедура смешивания может объединить их перед выводом. Lo-Fi Motion использует это для переходов — плавная смена между двумя эффектами путём интерполяции значений буферов.

Буфер также позволяет режим полузнакового разрешения. Буфер 32x48 отображается на экран двумя записями атрибутов на знакочечку (одна для “верхней половины” и одна для “нижней половины”), используя трюк с таймингом — перезапись атрибутов на середине строки развёртки. Это удваивает вертикальное разрешение ценой более сложного кода вывода и более жёстких ограничений тайминга.

Сами эффекты

Каждый эффект — вариация на темы из предыдущих глав: **plasma** (сумма синусоид из Главы 9), **rotozoomer** (обход текстуры из Главы 7), **fire** (клеточный автомат, усредняющий соседей), **rain** (система частиц) и **bigpic** (предварительно сжатая растровая анимация, распаковываемая покадрово, с использованием техник из Главы 14). Ни один из этих эффектов не нов. Суть в том, что при разрешении атрибутов каждый из них достаточно дёшев, чтобы четырнадцать уместились в одном демо, построенном за две недели. Решение lo-fi — мультиплексор силы.

Тулчейн

Тулчейн restorer'a для Lo-Fi Motion — конкретный ответ на вопрос “какие инструменты нужны для создания демо?”

Ассемблер: sjasmplus. Стандартный Z80-макроассемблер для современной ZX-сцены. Банкинг памяти (директивы SLOT/PAGE), условная компиляция, макросы, INCBIN для встроенных данных, DISPLAY для диагностики при сборке, вывод в .tap/.sna/.trd. Таблица сцен, код эффектов, сжатые данные и движок компилируются одним вызовом sjasmplus.

Эмулятор: zemu. Эмулятор, выбранный restorer'ом для Lo-Fi Motion. Unreal Speccy и Fuse одинаково распространены. Что важно — точный тайминг и быстрая перезагрузка: нужно тестировать новую сборку каждые несколько минут.

Графика: BGE 3.05 + Photoshop. BGE (Burial Graphics Editor, автор Sinn/Delirium Tremens) — графический редактор для ZX Spectrum, широко используемый в русской сцене для создания атрибутной графики непосредственно на целевой платформе. Пререндеренные PC-изображения проходят через Photoshop (или Multipaint, GIMP) и пользовательские скрипты.

Скрипты: Ruby. Автоматизация конвейера преобразования: изображения в данные атрибутов, таблицы синусов в бинарные инклуды, анимационные

последовательности в дельта-сжатые потоки. Python, Perl и Processing одинаково распространены. Что важно — преобразование автоматизировано и воспроизводимо.

Сжатие: `hrust1opt`. Hrust 1 с оптимальным парсингом. Z80-распаковщик релоцируемый (использует стек как рабочий буфер), удобен для демо, которые подключают и отключают данные в банках памяти.

Практический урок: нет единственно “правильного” тулчайна. Правильный — тот, где каждый шаг от исходного ассета до финального бинарника автоматизирован, изменение одного входа регенерирует все зависимые выходы, и вся сборка завершается за секунды. Любой ручной шаг — это баг, ждущий своего часа в 2 часа ночи перед дедлайном компо.

Конвейер сборки

Инструменты соединяются через **Makefile** (или аналогичный скрипт сборки). Конвейер для Lo-Fi Motion выглядит примерно так:

```

|
v
Ruby conversion scripts
|
v
Binary includes (.bin, .hru)
|
v
sjasmplus assembly
|
v
Output binary (.trd or .sna)
|
v
Test in emulator (zemu)
```

Каждая стрелка — правило в Makefile. Поменяй PNG, запусти `make`, и вся цепочка выполнится заново — преобразование, сжатие, сборка — порождая свежий бинарник за секунды. Загрузи его в эмулятор, посмотри результат, реши, что менять, отредактируй исходник, запусти `make` снова. Этот цикл редактирование-сборка-тестирование, измеряемый в секундах, — то, что позволяет построить четырнадцать эффектов за две недели.

Makefile также служит документацией. Чтение правил сборки говорит тебе, какие именно скрипты порождают какие выходные файлы, от каких файлов данных зависят какие эффекты, и как выглядит полный график зависимостей. Когда ты возвращаешься к проекту после шестимесячного перерыва, Makefile рассказывает, как всё соединяется.

Временная шкала: две недели вечеров

Lo-Fi Motion был построен примерно за две недели вечерних сеансов кодинга. restorer работал на основной работе. Вечера были единственным доступным временем.

Эта времененная шкала реалистична для lo-fi-атрибутного демо и поучительна для любого, кто планирует свою первую продукцию. Разбивка выглядит примерно так:

- **Дни 1-2:** Архитектура движка. Система таблицы сцен, виртуальный буфер, процедура вывода, базовый фреймворк. Запустить один эффект (plasma) через полный конвейер.
- **Дни 3-7:** Эффекты. Два-три эффекта за вечер, когда фреймворк готов. Каждый эффект — 100-300 строк ассемблера, рендерящих в виртуальный буфер. Тестировать каждый по отдельности.
- **Дни 8-10:** Контент. Пререндеренные изображения, данные шрифтов, скрипты преобразования. Здесь скрипты на Ruby оправдывают себя.
- **Дни 11-12:** Интеграция. Все эффекты в таблице сцен, тайминг подогнан под музыку, переходы настроены. Здесь окупается рабочий процесс таблицы сцен с редактированием и пересборкой.
- **Дни 13-14:** Полировка и отладка. Цвета бордюра для визуализации тайминга (Глава 1), исправление эффектов, ломающихся на граничных слу-чаях, финальный проход сжатия для уместивания всего в память.

Ключевое наблюдение: движок и конвейер съедают первые два дня. Каждый последующий день выигрывает от этих инвестиций. Если пропустить работу над конвейером и захардкодить первый эффект прямо в экранную память, ты сэкономишь день авансом и потеряешь неделю позже, когда попытаешься добавить второй эффект и обнаружишь, что ничего не модульно.

20.3 Культура making-of

ZX Spectrum-демосцена имеет сильную культуру документирования того, как создаются демо. Это не универсально для демосцены в целом — на многих платформах демо выходят без документации, кроме титров. В Spectrum-сцене подробные статьи making-of — традиция, и Hype (hype.retroscene.org) — основная площадка для их публикации.

Eager: технический NFO

Когда Introspec выпустил Eager (to live) на ZBM Open Air 2015, ZIP-файл содержал file_id.diz — традиционный информационный файл демо — который пошёл далеко за пределы титров и приветствий. Это был технический текст: подход к атрибутному туннелю, оптимизация четырёхкратной симметрией, гибридная техника цифровых барабанов, архитектура асинхронной генерации кадров. Kylearan, рецензируя демо на Pouet, написал: “Огромное спасибо за один только nfo-файл, обожаю читать технические разборы! Помогает понять, что я вижу/слышу.”

Затем Introspec опубликовал ещё более подробную статью making-of на Hype, которая стала основным источником для Глав 9 и 12 этой книги. Статья объясняла не только что делает демо, но почему — обоснование каждого технического решения, ограничения, определившие архитектуру, творческие цели, сформировавшие визуальный дизайн.

Этот уровень документирования служит нескольким целям. Для зрителя он

углубляет понимание — знание того, как работает эффект, делает просмотр более, а не менее вознаграждающим. Для других кодеров это образование — статьи *making-of* на Hure — ближайший аналог технической учебной программы в ZX-сцене. Для автора это форма завершения — формулирование решений заставляет тебя понять собственную работу, а обратная связь сообщества (комментарии на Hure могут насчитывать сотни сообщений) подвергает твои рассуждения стресс-тестированию.

GABBA: другой рабочий процесс

Статья *diver4d making-of* для GABBA (2019) документирует радикально отличный рабочий процесс от Eager. Там, где Introspec потратил недели на скриптовый движок и асинхронный буфер кадров, *diver4d* использовал Luma Fusion — iOS-видеоредактор — как свой инструмент синхронизации.

Мы рассмотрели технические детали в Главе 12. Важен рабочий процесс: *diver4d* осознал, что покадровая аудиовизуальная синхронизация — это задача видеомонтажа, а не *программирования*. Выполняя работу по синхронизации в инструменте, предназначенном для этого, он мог итерировать тайминг за секунды вместо минут. Z80-код был слоем реализации; творческие решения принимались в видеоредакторе.

Это общий принцип. Рабочий процесс создания демо — не о том, чтобы делать всё на ассемблере. А о том, чтобы использовать правильный инструмент для каждой задачи. Ассемблер для внутренних циклов. Processing или Ruby для генерации кода. Photoshop или Multipaint для графики. Видеоредактор для тайминга. Makefile, чтобы связать всё вместе. Демо — результат; инструменты — то, что доведёт тебя до него быстрее всего.

NHBF: головоломка

Статья *making-of UriS* для NHBF (2025) документирует рабочий процесс на противоположном конце от конвейера Lo-Fi Motion с четырнадцатью эффектами. NHBF — это 256-байтное интро: вся программа, код и данные, помещается в меньшее пространство, чем один кадр атрибутов. “Рабочий процесс” — это один человек, уставившийся в hex-дамп, постоянно переставляющий инструкции в поисках более коротких кодировок, обнаруживающий, что значения регистров от одной процедуры случайно совпадают с потребностями в данных другой.

Мы рассмотрели конкретные техники в Главе 13. Урок для рабочего процесса — о творчестве, управляемом ограничениями. UriS описывает процесс как “игру в головоломки” — подходящая метафора, потому что пространство оптимизации в 256-байтном кодинге комбинаторно. Ты не можешь спланировать путь к решению. Ты можешь только продолжать переставлять кусочки и быть внимательным к случайным совпадениям. Открытие Art-Tор, что значения регистров от процедуры очистки экрана совпали с длиной текстовой строки, не было запланировано. Оно было замечено.

Это важно для рабочего процесса создания демо любого масштаба. Даже в полноразмерном демо с нормальным движком, Makefile и таблицей сцен бывают моменты, когда лучшее решение приходит от того, что ты отступаешь и смотришь на всю картину целиком, замечая случайное совпадение между двумя

системами, разработанными независимо. Головоломочный образ мышления не эксклюзивен для sizecoding. Это режим мышления, который улучшает всю работу над демо.

20.4 Тулчейн в деталях

ZX Spectrum-демосценический тулчейн сошёлся на стандартном наборе. Вот типичная структура проекта:

```

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

```

Ассемблер: sjasmplus

Рабочая лошадка. Банкинг памяти через директивы SLOT/PAGE, условная компиляция, макросы, INCBIN для встроенных данных, DISPLAY для диагностики при сборке и вывод в .tap/.sna/.trd. Типичное демо компилируется одним вызовом sjasmplus.

Эмуляторы

Unreal Speccy предпочтается многими демосценерами русской сцены за детерминированный тайминг и точную эмуляцию Pentagon, с поддержкой TR-DOS, TurboSound и нескольких моделей клонов. **Fuse** широко доступен на Linux и macOS. **zemу** — ещё один вариант, используемый restorer'ом для Lo-Fi Motion. Для отладки на уровне исходного кода **DeZog** в VS Code подключается к ZEsarUX и предоставляет точки останова, инспекцию регистров и просмотр памяти.

Выбери один эмулятор для основной разработки. Тестируй на других перед релизом. Демо, которые работают в одном эмуляторе и падают в другом — партийная традиция, которой лучше избегать.

Графика и генерация кода

Multipaint принудительно обеспечивает ограничения атрибутов в реальном времени — специально создан для 8-битного пиксель-арта. **Photoshop**, **GIMP** или **Aseprite** предлагают творческую свободу, но требуют скриптов преобразования (Python, Ruby, Processing) для квантизации и экспорта. **Processing** обрабатывает генеративную графику и генерацию кода — Introspec использовал его для генерации развёрнутых последовательностей кода хаос-зумера (Глава 9).

Автоматизация сборки и CI

Makefile должен автоматизировать полный конвейер: от исходных ассетов через скрипты преобразования до сжатия и сборки. Если любой шаг требует ручного вмешательства, он подведёт в 2 часа ночи перед дедлайном.

CI через GitHub Actions становится всё более распространённым. Рабочий процесс, который собирает при каждом пуше, ловит неявные зависимости — демо собирается на твоей машине, но падает в чистом окружении из-за необъявленной версии инструмента. Исходники Lo-Fi Motion на GitHub, опубликованы как эталонная реализация: клонируй, запусти `make`, получи рабочий бинарник. Эта открытость необычна для демосцены и ценна для обучения.

Синхронизация и композитинг

Самая сложная часть демо — не эффекты, а *тайминг*. Когда запустить плазму. Когда переключиться на скроллер. Какой удар бита вызывает цветовую вспышку. Это — синхронизация, и ZX Spectrum-сцена выработала многоуровневый подход, сочетающий инструменты из демосцены с универсальными видеоредакторами.

Таблица синхронизации. На уровне Z80 синхронизация — это таблица данных:

```
dw 0,      effect_logo      ; frame 0: show logo
dw 150,    effect_plasma   ; frame 150: start plasma
dw 312,    flash_border     ; frame 312: beat hit, flash
dw 500,    effect_scroll   ; frame 500: start scroller
dw 0          ; end marker
```

Движок увеличивает счётчик кадров на каждом VBlank, сравнивает его со следующей записью в таблице и вызывает обработчик, когда наступает нужный кадр. Это простейший возможный механизм синхронизации. И это именно то, что в конечном счёте выполняет каждое ZX Spectrum-демо — вне зависимости от того, как были определены эти номера кадров.

Вопрос в том: как *найти* правильные номера кадров? Существует пять подходов, от простейшего к самому изощрённому. (Приложение J подробно описывает полные рабочие процессы, конвейеры экспорта и пошаговые рецепты для каждого инструмента.)

Подход 1: Vortex Tracker + ручной тайминг. Открой свой .pt3 в Vortex Tracker II. В правом нижнем углу отображается текущая позиция (паттерн, строка, кадр). Проиграй мелодию, записывай номера кадров, где приходятся

удары, акценты и переходы между фразами. Впиши их в таблицу синхронизации. Пересобери, проверь, подкорректируй. Именно так работает большинство ZX-демосценеров, включая Kolnogorov (Vein): «Vortex + видеоредактор. В Vortex в правом нижнем углу показан кадр — я смотрел, к каким кадрам привязываться, создавал таблицу с записями dw frame, action и синхронизировал по ней.»

Преимущество: ты одновременно слышишь музыку и видишь номера кадров. Недостаток: итерирование медленное — каждое изменение требует пересборки демо и просмотра с самого начала.

Подход 2: Видеоредактор как планировщик синхронизации. Рабочий процесс diver4d для GABBA исходил из того, что покадровая синхронизация — это задача видеомонтажа. Запиши каждый эффект как видеоклип, импортируй клипы и музыку в видеоредактор (DaVinci Resolve, Blender VSE), перемотай до идеальных точек склейки и считай номера кадров. Kolnogorov: «Я экспортировал клипы с эффектами в видео, собрал их в видеоредакторе, наложил музыкальную дорожку и смотрел, в каком порядке эффекты смотрятся лучше всего, записывая кадры, где должны происходить события.» Ключевое слово — *смотрел*: это визуальный, интуитивный процесс. (Приложение J.2–J.3 подробно описывает Blender VSE, DaVinci Resolve и рабочий процесс GABBA.)

Подход 3: GNU Rocket. Стандартный инструмент синхронизации на PC- и Amiga-демосценах — трекер-подобный редактор, где столбцы — именованные параметры, а строки — временные шаги. Ты расставляешь ключевые кадры с интерполяцией (step, linear, smooth, ramp) и редактируешь в реальном времени, пока демо работает через TCP. Z80-клиент непрактичен, но рабочий процесс переносится: спроектируй кривые синхронизации в Rocket, экспортируй ключевые кадры, преобразуй в Z80-таблицы dw/db Python-скриптом. (Приложение J.2 описывает полный конвейер Rocket → Z80; Приложение J.7 даёт пошаговый рецепт.)

Подход 4: Blender для пре-визуализации. Для сложных демо — раскидай эффекты в виде цветных полосок на таймлайне VSE с музыкальной дорожкой, анимируй параметры-заглушки в Graph Editor, затем экспортируй номера кадров и значения ключевых кадров через Python API Blender непосредственно как данные, готовые для Z80. (Приложение J.2–J.3 описывает рабочие процессы и для VSE, и для Graph Editor.)

Подход 5: Игровые движки как генераторы данных. Unity и Unreal избыточны в качестве движков демо, но идеальны как генераторы данных: VR-захват движения (рисуй траектории контроллером), GPU-симуляция частиц (экспортируй позиции по кадрам) и прототипирование шейдеров (итерируй алгоритм на полной скорости, затем переводи на Z80). Blender покрывает большую часть этого для не-VR задач. Конвейер экспорта всегда один и тот же: float → 8-bit fixed-point → дельта-кодирование → транспонирование → сжатие → INCBIN. (Приложение J.4 описывает полный конвейер со сравнительными таблицами и пошаговым рецептом VR-захвата.)

На PC-демосцене существует параллельная экосистема инструментов для создания демо, построенных на той же философии процедурной генерации и экстремального сжатия: Werkzeug/kkrunchy от Farbrausch (открыт в 2012), TiXL (визуальный моушен-дизайн на нотах, MIT), Bonzomatic (лайв-кодинг шейдеров) и музыкальные синтеза-

торы вроде Sointu и WaveSabre. Ни один из них не нацелен непосредственно на Z80, но мышление идентичное — ZX Spectrum-эквивалент нодового графа Werkzeug — это твой Python-скрипт сборки, генерирующий таблицы подстановки и выдающий директивы INCBIN. Приложение J.5 описывает историю, а Приложение J.6 обзорно рассматривает музыкальные инструменты, включая Furnace — современный трекер с прямой поддержкой AY-3-8910.

FIGURE: Vortex Tracker II – frame counter

VT2 main window with pattern editor visible.
Bottom-right: position display showing pattern:row and absolute frame number.
Highlight/circle the frame counter.

Caption: "The frame number in VT2's status bar maps directly to the PT3 player's interrupt counter on the Spectrum. What you see here is what your sync table references."

Screenshot needed: open any .pt3 in VTI fork, play to a mid-song position, capture with frame number visible.

См. Приложение J для псевдо-скриншотов GNU Rocket, Blender VSE, Blender Graph Editor и TiXL, а также подробных описаний инструментов и пяти пошаговых рецептов экспорта.

Человеческий фактор. Kolnogorov формулирует принцип, который все опытные демосценеры понимают, но редко озвучивают: «Даже если мы знаем, что малый барабан бьёт каждые 16 нот, и мы мигаем бордюром каждые 16 нот — это будет выглядеть мёртво и механически. Суть синхронизации в том, что она должна быть намеренно неравномерной и местами ломаной.»

Алгоритмическая синхронизация — срабатывание на каждый удар, затухание на каждой границе фразы — ощущается механической. Лучшая синхронизация в демо следует музыкальным фразам, а не отдельным ударам. Некоторые события срабатывают чуть раньше удара (нагнетая напряжение). Некоторые — после (создавая неожиданность). На некоторых фразах визуальных изменений нет вовсе (создавая ожидание следующего удара). Вот почему ручные таблицы синхронизации, кропотливо собранные человеком, который смотрит и слушает, неизменно дают лучший результат, чем любая автоматическая система.

Практический вывод: даже если ты используешь Rocket или Blender для планирования синхронизации, финальный проход всегда ручной. Смотри демо с музыкой. Корректируй номера кадров на слух. Добавляй смещённые удары и намеренные паузы, которые делают синхронизацию живой.

20.5 Культура компо

Демо без компо — это видео на YouTube. Демо на компо — это перформанс: показанное на большом экране, перед зрителями, в сравнении с другими

работами, с призами на кону. Компо — место, где работа встречает свою аудиторию, и культура вокруг компо формирует работу.

Основные пати

ZX Spectrum-демосцена обслуживается несколькими регулярными пати, каждая со своим характером.

Chaos Constructions (CC) — крупнейшее и наиболее престижное ZX-мероприятие, проводимое в Санкт-Петербурге. ZX-демокомпо на СС привлекает сильнейшие работы: Break Space (2016), продолжатели Eager и продукции от групп Thesuper, 4D+TBK и Placeholders. СС — место, куда идут соревноваться на высшем уровне. Аудитория большая, знающая и беспощадная.

DiHalt проводится в Нижнем Новгороде и имеет как летнее мероприятие, так и зимнюю "Lite"-версию. DiHalt, как правило, экспериментальнее СС — аудитория приветлива к дебютантам, и атмосфера поощряет риск. Lo-Fi Motion был выпущен на DiHalt 2020. Если ты участвуешь в своём первом компо, DiHalt Lite — хороший выбор.

Multimatograf — меньшее мероприятие с традицией поощрения новых работ. Категории компо широки, требования к участникам минимальны, и атмосфера поддерживающая. Introspec рецензировал компо Multimatograf на Нуре, иногда критически — он предъявляет к каждой пати одинаковые стандарты — но само мероприятие приветливо к новичкам.

CAFe (Creative Art Festival) — мероприятие демосцены с более широким охватом (не исключительно ZX), но ZX-категории привлекают сильные работы. GABBA заняла первое место на CAFe 2019.

Revision — крупнейшее мероприятие демосцены в мире, проводимое ежегодно в Саарбрюккене, Германия. Оно не специализировано на ZX, но категории "8-bit demo" и "oldschool" принимают работы для ZX Spectrum. Соревнование на Revision означает показ твоей работы глобальной демосцене — аудитории из тысяч людей, большинство из которых никогда не видели демо для Spectrum. Megademica от SerzhSoft выиграла компо 4K intro на Revision 2019, доказав, что ZX-работы могут конкурировать на глобальной сцене.

Как участвовать в первом компо

Процесс менее пугающий, чем кажется.

1. Выбери пати. Начни с меньшего мероприятия — DiHalt Lite, Multimatograf или локальная пати, если она есть в твоём регионе. На крупных пати ожидания выше, и давление конкуренции с опытными группами на СС может быть контрпродуктивным для первого участия.

2. Знай правила. Каждая пати публикует правила компо, определяющие: требования к платформе (какая модель Spectrum, какая конфигурация эмулятора), формат файла (.tap, .trd, .sna), максимальный размер файла, принимаются ли удалённые работы, и дедлайны подачи. Прочитай правила. Следуй правилам. Технически впечатляющее демо, поданное как .tzx, когда правила требуют .trd, будет дисквалифицировано.

3. Тестируй на целевой платформе. Если пати запускает работы на реальном железе (физический Pentagon или Scorpion), тестируй на этом железе или на эмуляторе, настроенном на соответствие. Демо, которые прекрасно работают на одной модели и падают на другой — удручающее частое явление. Различия тонкие: тайминг спорной памяти, задержки переключения банков, особенности чипа AY. Глава 15 покрывает машинно-специфичные детали; глава 5 серии “GO WEST” Introspec’а покрывает подводные камни переносимости.

4. Подавай рано. Большинство пати принимают удалённые работы по электронной почте или через веб-форму. Подай на день раньше, если возможно. Подачи в последний момент стрессовые и подверженные ошибкам (загрузка не того файла, забытый обязательный файл метаданных). Пати-версия может быть несовершенной — многие демо обновляются до “финальных” версий после пати, исправляя баги, обнаруженные во время показа на компо.

5. Напиши file_id.diz или NFO. Включи текстовый файл с титрами (кто что делал), требованиями к платформе (какая модель, какой режим) и — если ты готов — кратким техническим описанием. Зрители ценят знание того, на что они смотрят. Сцена ценит документацию. И ты оценишь, что написал это, когда через три года попытаешься вспомнить, как работает генерация таблицы plasma.

6. Смотри компо. Если ты на пати лично, смотри своё демо на большом экране вместе с аудиторией. Опыт видения своей работы, показанной публично, слышания реакции зрителей, сравнения своей работы с другими — вот ради чего существуют компо. Если ты подаёшь удалённо, смотри стрим, если он доступен. Некоторые пати публикуют записи компо на YouTube позже.

7. Не ожидай победы. Твоё первое участие — учебный опыт. Цель — закончить что-то, подать и увидеть показанным. Место — бонус. Обратная связь, которую ты получишь — от зрителей, от других сценеров, от собственной реакции на видение работы на большом экране — стоит больше любого приза.

Удалённые работы принимаются на большинстве ZX-мероприятий. Lo-Fi Motion был удалённой работой на DiHalt 2020. Некоторые пати проводят исключительно онлайн-мероприятия, транслируемые на YouTube или Twitch. Если ближайшее демосценическое мероприятие в 12 часах лёта, онлайн-компо — жизнеспособная отправная точка.

20.6 Сообщество

ZX Spectrum-демосцена достаточно мала, чтобы большинство активных участников знали друг друга, и достаточно велика, чтобы поддерживать несколько активных сообществ.

Hype (hype.retroscene.org)

Основной русскоязычный форум для обсуждения ZX Spectrum-демосцены. Основан и модерируется Introspec’ом, здесь публикуются статьи making-of, технические руководства, обзоры компо и дискуссии о дизайне, составляющие основной материал для этой книги. Темы набирают сотни комментариев, опытные кодеры детально обсуждают подсчёт тактов. Для не говорящих по-русски

инструменты машинного перевода в браузере достаточно хорошо справляются с прозой, а Z80-ассемблер читается одинаково на любом алфавите.

Культура прямая. Если ты постишь демо с ошибкой тайминга, кто-то скажет тебе точно, какой такт неправильный. Эта прямота порождает подлинные технические дискуссии, а не вежливые, но бесполезные поощрения.

ZXArt (zxart.ee)

Всеобъемлющий архив творческих работ для ZX Spectrum — демо, музыки, графики, игр, журналов и метаданных. Каждую продукцию в этой книге можно найти на ZXArt со скриншотами, титрами, результатами пати и загрузками. ZXArt также хранит оцифрованные ZX-журналы в формате TRD (Spectrum Expert, Born Dead, ZX Format), содержащие оригинальные статьи, заложившие техники, которым учит эта книга.

Pouet (pouet.net)

Глобальная база данных продукции демосцены. Для ZX-сцены Pouet — мост к более широкому сообществу: ZX-демо оцениваются людьми, которые преимущественно смотрят PC- или Amiga-продукции. Смена перспективы ценна: технически блестящий внутренний цикл, впечатляющий читателей Нуре, может быть невидим для комментатора Pouet, который фокусируется на визуальном воздействии и музыкальной синхронизации. Pouet также хранит NFO-файлы — когда не можешь найти статью making-of на Нуре, проверь NFO на Pouet.

20.7 Управление проектом для создателей демо

Создание демо — это управление проектом. У проекта есть дедлайн (дата пати), результаты (финальный бинарник), зависимости (музыка, графика, эффекты, движок) и обычно команда участников с конкурирующими приоритетами. Управление этим не гламурно, но это то, что отделяет завершённые демо от заброшенных прототипов.

Минимально жизнеспособное демо

Начни с простейшей возможной версии демо, которая является завершённой — не отполированной, не впечатляющей, но завершённой. Один эффект, одна мелодия, нормальное начало и конец. Запусти это через полный конвейер сборки в первые несколько дней. Это твоя страховочная сеть. Если всё пойдёт не так — если сложный эффект не работает, если музыкант опаздывает с финальным треком, если твой жёсткий диск умрёт за неделю до пати — у тебя есть что подать.

Затем итерируй. Добавляй эффекты по одному. Замени placeholder-музыку, когда прибудет финальный трек. Добавь переходы, отполируй тайминг, оптимизируй использование памяти. Каждая итерация порождает завершённое, подаваемое демо, которое лучше предыдущего. В любой момент ты можешь остановиться и подать то, что есть.

Именно так был построен Lo-Fi Motion. restorer не написал четырнадцать эффектов и потом сшил их вместе. Он построил движок и один эффект, проверил, что они работают, затем добавлял эффекты по одному. Каждый вечер работы порождал чуть лучшее демо. Если бы у него кончилось время на десяти эффектах вместо четырнадцати, демо всё равно было бы завершённым и подаваемым.

Работа с соавторами

Большинство демо — коллаборации. Три принципа держат их на ходу:

Установи формат данных рано. Музыканту нужно знать: PT3 или пользовательский плеер? Один AY или TurboSound? Как сигнализируются триггеры барабанов? Художнику нужно знать: разрешение атрибутов или пиксельное? Ограничения цвета? Максимальный размер файла? Получить TurboSound-композицию, когда твой движок поддерживает только один AY — катастрофа, и это твоя вина за то, что не указал ограничения.

Сообщай временную шкалу. Если пати через четыре недели, скажи музыканту, что тебе нужен трек через две. Запас — для интеграции, отладки и сюрпризов.

Предоставляй placeholder'ы. Используй placeholder-.pt3 с правильным темпом до прибытия финального трека. Используй программистскую графику до прибытия финальной. Движок не должен зависеть от финальных ассетов. Когда настоящие ассеты прибудут, вставь их в конвейер и пересобери.

Отладка и тестирование

Баги в демо уникально болезненны, потому что проявляются перед аудиторией. Краш во время показа на компо — и техническая неудача, и социальный конфуз. Тестирование не опционально.

Тестируй на нескольких эмуляторах. Каждый эмулятор имеет немного различный тайминг, инициализацию памяти и поведение AY. Демо, которое работает в Unreal Speccy, но падает в Fuse, вероятно, имеет допущение о тайминге или памяти, верное для Pentagon, но не для стандартного Spectrum.

Тестируй с холодного старта. Очисти всю память перед загрузкой демо. Не предполагай никаких значений регистров или содержимого памяти от предыдущей программы. Если твоё демо работает после запуска предыдущего демо, но падает с чистой загрузки, у тебя баг инициализации.

Тестируй файл для компо, а не девелоперский бинарник. Файл, который ты подаёшь, должен быть точно тем файлом, который ты тестировал. Не “быстро перекомпилированной” версией с исправлением в последний момент. Исправления в последний момент вносят баги в последний момент.

Используй цвета бордюра для тайминга. Техника из Главы 1: устанавливай бордюр в разные цвета в разных точках цикла кадра. Если цветная полоска бордюра заходит в видимую область, твой код слишком медленный. Если нет — у тебя запас. Это самый быстрый способ проверить, что эффект укладывается в бюджет кадра.

20.8 Преодоление платформы: “MORE” Introspec’а

В феврале 2015 Introspec опубликовал на Нуре короткое эссе, озаглавленное просто “MORE”. Это не техническая статья. В ней нет кода, нет подсчёта тактов, нет внутренних циклов. Это вызов ZX Spectrum-сцене — и, по расширению, всем, кто работает в рамках аппаратных ограничений.

Аргумент в том, что лучшие демо — это не те, которые делают самые впечатляющие вещи вопреки ограничениям платформы. Это те, которые полностью преодолевают платформу — которые создают впечатления, значимые на любом железе. Ограничения платформы формируют технику, но не должны ограничивать амбицию.

Двух пикселей достаточно, чтобы рассказать историю.

Это самая цитируемая строка Introspec’а. Она означает: художественное содержание демо не определяется его разрешением, глубиной цвета, количеством полигонов или частотой дискретизации. Два пикселя — две ячейки атрибутов, две точки на сетке 32x24 — могут рассказать историю, если тайминг правильный, контекст ясен и намерение искренно. Технология служит искусству, а не наоборот.

Introspec ссылается на работу Джеймса Хьюстона “Big Ideas (Don’t Get Any)” — видео, где Sinclair ZX Spectrum, матричный принтер и другое устаревшее оборудование исполняют песню Radiohead. Проект трогает не из-за технического достижения, а потому что выбор оборудования *что-то значит*. Устаревание — суть. Хрупкость — красота.

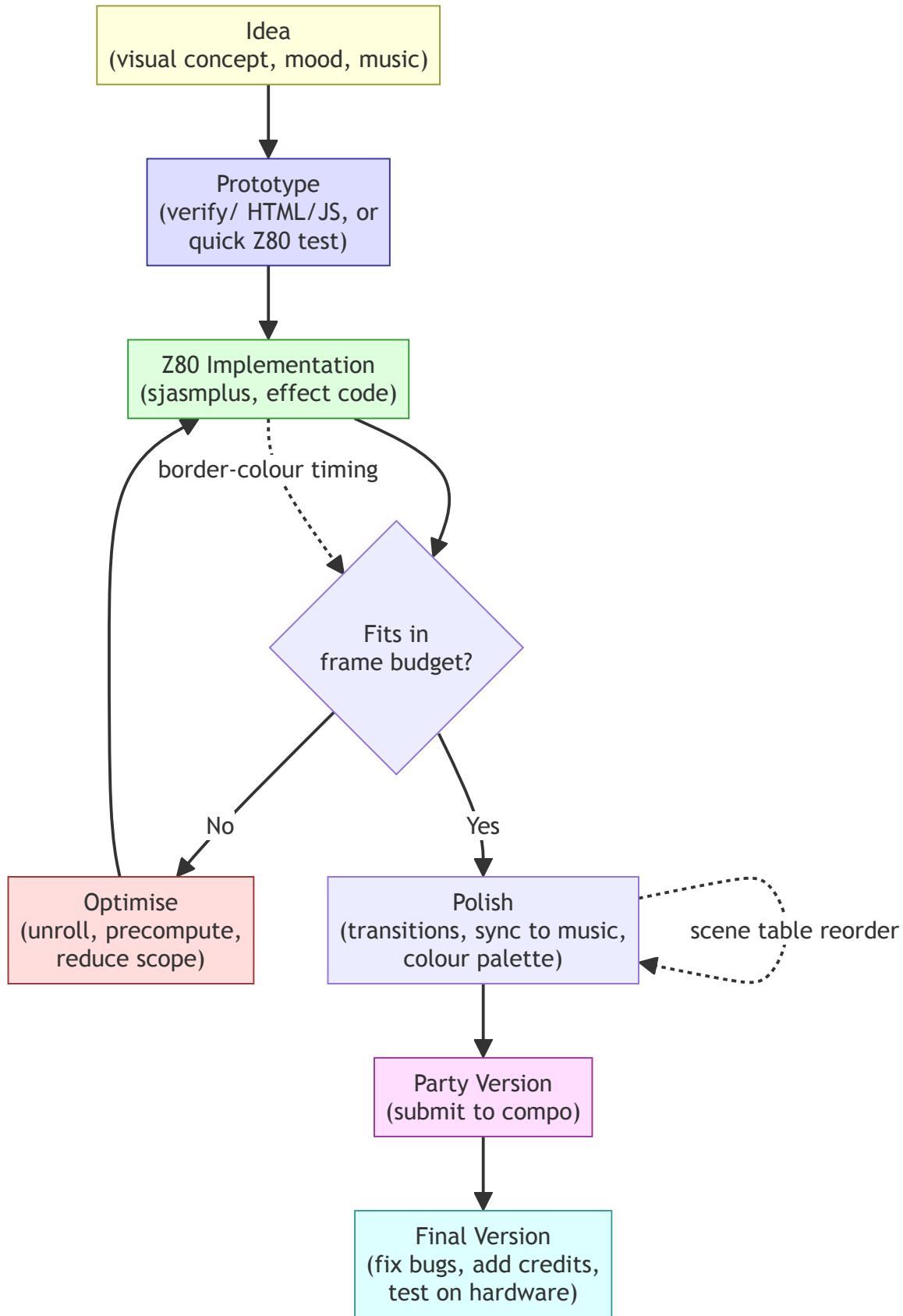
Практическое следствие: техника необходима, но недостаточна. Ты можешь овладеть каждым эффектом в этой книге и всё равно произвести демо, которое никто не запомнит. Что делает демо запоминающимся — не то, что оно делает, а то, что оно говорит. Даже абстрактное демо имеет личность: его ритм говорит что-то о напряжении и разрядке; его цветовая палитра вызывает настроение; его выбор музыки создаёт эмоциональный контекст. Кодер, который относится к этому как к мелочам, производит техдемо. Кодер, который относится к этому как к дизайнерским решениям, производит демо.

Lo-Fi Motion принял свою lo-fi-эстетику как идентичность. Eager превратил сетку 32x24 из ограничения в творческий выбор. NHBF нашёл красоту в головоломке 256 байт. В каждом случае ограничение стало медиумом.

Вот чего требует “MORE”. Не больше полигонов, не больше цветов, не больше эффектов. Больше амбиций. Больше намерения. Больше готовности относиться к демо для ZX Spectrum как к форме искусства.

20.9 Твоё первое демо: практическая дорожная карта

Для читателя, следовавшего этой книге с Главы 1 и желающего сделать демо, вот конкретный путь.



Итеративный цикл: На пути от реализации к проверке тайминга и обратно проходит большая часть времени разработки. Этап про-

тотипирования (HTML/JS или быстрый набросок на Z80) проверяет визуальную концепцию до перехода к полной реализации. Таблица сцен позволяет тривиально менять порядок эффектов на этапе полировки.



Рис. 40: Фреймворк демо со слотами эффектов, управляемыми таблицей сцен; движок циклически переключает несколько визуальных эффектов

Неделя 1: Фундамент

- Настрой тулчейн.** Установи sjasmplus, выбери эмулятор (Unreal Speccy, Fuse или ZEsarUX), настрой директорию проекта с Makefile. Проверь, что ты можешь собрать минимальную программу и запустить её в эмуляторе.
- Построй движок таблицы сцен.** Напиши минимальный движок, который читает таблицу сцен и вызывает процедуры эффектов на указанную длительность. Начни с архитектуры Lo-Fi Motion: номер банка, адрес входа, количество кадров. Заставь это работать с одним фиктивным эффектом (залить экран цветом, инкрементировать цвет каждый кадр).
- Добавь музыку.** Интегрируй PT3-плеер в обработчик прерывания IM2 (Глава 11). Подставь любой .pt3-файл как placeholder. Проверь, что музыка играет, пока фиктивный эффект работает.

Неделя 2: Эффекты

- Построй свой первый настоящий эффект.** Атрибутная plasma — естественная отправная точка: она дешёвая, визуально богатая и хорошо изученная (Глава 9). Рендири в виртуальный буфер и копирай в память атрибутов.
- Построй свой второй эффект.** Fire, rotozoomer, rain, цветные полосы — выбери один из эффектов, рассмотренных в Части II. Два эффекта и переход между ними составляют минимальное демо.
- Добавь переход.** Простой crossfade между двумя атрибутными буферами: интерполяция значений цвета за 25–50 кадров. Или жёсткая склейка, синхронизированная с битом в музыке.

Неделя 3: Полировка

7. **Замени placeholder-музыку.** Если у тебя есть музыкант-соавтор, интегрируй финальный трек. Если нет, потрать время на выбор .pt3, который подходит настроению и ритму твоего демо.
8. **Настрой тайминг.** Здесь таблица сцен оправдывает себя. Переставь эффекты, подгони длительности, привяжи переходы к музыкальным событиям. Пересобирай и тестируй многократно.
9. **Добавь начало и конец.** Экран загрузки (сжатый ZX0, Глава 14), титульная заставка, экран финальных титров. Первое и последнее впечатления важны.

Неделя 4: Релиз

10. **Тестируй.** Несколько эмуляторов. Холодный старт. Точно тот файл, который будешь подавать.
11. **Напиши NFO.** Титры, требования к платформе, приветствия и — если ты щедр — техническое описание того, как работают эффекты. Будущий ты будет благодарен.
12. **Подавай.** Выбери пати. Следуй правилам. Загрузи файл. Затем смотри компо и наслаждайся видением своей работы на экране.

Твоя первая работа вряд ли займёт призовое место. Отнесись к ней как к учебному опыту: обратная связь от просмотра своей работы на большом экране и сравнения её с другими работами ценнее любого приза. Каждое следующее демо будет лучше, потому что ты будешь знать, что исправить.

Итого

- **Дизайн — это всё.** Introspec определяет дизайн демо как “совокупность всех компонентов демо, как видимых, так и скрытых” — архитектуру кода, раскладку памяти, ритм и эмоциональную дугу, а не только визуальные эффекты.
- **Lo-Fi Motion предоставляет воспроизводимый шаблон производства:** таблица сцен управляет структурой демо, эффекты рендерят в виртуальные буферы с одним байтом на пиксель, а тулчейн (sjasmplus + zemu + скрипты Ruby + hrust1opt) соединяется через Makefile. Четырнадцать эффектов были построены за две недели вечерней работы.
- **Паттерн таблицы сцен** отделяет контент от движка. Добавление, удаление или перестановка эффектов означает редактирование таблицы данных, а не перестройку кода. Это поддерживает быструю итерацию ритма и структуры.
- **Культура making-of — сила ZX-сцены.** Подробные технические разборы — от NFO Eager до рабочего процесса GABBA с видеоредактором и 256-байтной головоломки NHBF — служат образованием, документацией и строительством сообщества.

- **Стандартный тулчейн** сходится на sjasmplus (ассемблер), Unreal Speccy или Fuse (эмулятор), BGE или Multipaint (графика), скрипты Ruby или Python (преобразование и генерация кода), ZX0 или hrust1opt (сжатие) и Makefile (автоматизация сборки). CI через GitHub Actions становится всё более распространённым.
- **Синхронизация** — самая сложная часть демо. Многоуровневый подход: определи номера кадров в Vortex Tracker или видеоредакторе (DaVinci Resolve, Blender VSE), при необходимости спланируй интерполированные кривые параметров в GNU Rocket, экспортируй в Z80-таблицы dw frame, action. Финальный проход всегда ручной — алгоритмическая синхронизация ощущается роботизированной; расставленная вручную — следует фазам, а не ударам. (Приложение J описывает все инструменты синхронизации, конвейеры генерации данных и пошаговые рецепты экспорта.)
- **Культура компо** сосредоточена вокруг событий Chaos Constructions, DiHalt, Multimatograf, CAFe и Revision. Участие в первом компо требует выбора подходящего мероприятия, соблюдения правил, тщательного тестирования и ранней подачи.
- **Сообщество** живёт на Hype (технические дискуссии, статьи making-of), ZXArt (архив продукции) и Pouet (глобальная база данных демосцены).
- **Управление проектом имеет значение.** Сначала построй минимально жизнеспособное демо, затем итерируй. Установи форматы данных с соавторами рано. Тестируй на нескольких эмуляторах с холодного старта. Подавай точно тот файл, который тестировал.
- “**MORE**” Introspec’а призывает создателей демо преодолевать ограничения платформы: “Двух пикселей достаточно, чтобы рассказать историю.” Технология служит искусству, а не наоборот. Лучшие демо — не самые технически впечатляющие, а те, где каждый компонент, видимый и скрытый, служитциальному творческому видению.

Далее: Глава 21 — Полная игра: ZX Spectrum 128K. Мы переходим от демо к играм, интегрируя всё из Частей I-V в полноценный горизонтальный платформер.

Источники: restorer, “Making of Lo-Fi Motion”, Hype, 2020 (hype.retroscene.org/blog/demo/1023.html); Introspec, “О дизайне”, Hype, 2015 (hype.retroscene.org/blog/demo/64.html); Introspec, “MORE”, Hype, 2015 (hype.retroscene.org/blog/demo/87.html); Introspec, “Making of Eager”, Hype, 2015 (hype.retroscene.org/blog/demo/261.html); diver4d, “Making of GABBA”, Hype, 2019 (hype.retroscene.org/blog/demo/948.html); UriS, “NHBФ Making-of”, Hype, 2025 (hype.retroscene.org/blog/dev/1120.html)

Глава 21: Полная игра - ZX Spectrum 128K

«Единственный способ узнать, работает ли твой движок, – выпустить игру.»

У тебя есть спрайты (Глава 16). У тебя есть скроллинг (Глава 17). У тебя есть игровой цикл и система сущностей (Глава 18). У тебя есть столкновения, физика и ИИ врагов (Глава 19). У тебя есть музыка на AY и звуковые эффекты (Глава 11). У тебя есть сжатие (Глава 14). У тебя есть 128K банковской памяти, и ты знаешь, как обращаться к каждому её байту (Глава 15).

Теперь тебе нужно поместить всё это в один бинарный файл, который грузится с ленты, показывает экран загрузки, представляет меню, проводит пять уровней бокового скроллера с четырьмя типами врагов и боссом, ведёт таблицу рекордов и помещается в файл .tap.

Это глава интеграции. Здесь не появляется новых техник. Вместо этого мы сталкиваемся с проблемами, которые возникают только тогда, когда все подсистемы должны сосуществовать: конкуренция за память между графическими банками и кодом, бюджеты кадра, которые переполняются, когда скроллинг, спрайты, музыка и ИИ одновременно требуют свою долю, системы сборки, которые должны координировать десятки шагов конвертации данных, и тысяча мелких решений о том, куда разместить в 128K банковской памяти.

Игра, которую мы создаём, называется *Ironclaw* – пятиуровневый боковой платформер, в котором механический кот пробирается через серию всё более враждебных фабричных этажей. Жанр выбран намеренно: боковые платформеры требуют одновременной работы всех подсистем и не дают спрятаться. Если скроллинг заикается, ты это видишь. Если рендеринг спрайтов не укладывается в кадр, ты это чувствуешь. Если обнаружение столкновений даёт сбой, игрок проваливается сквозь пол. Платформер – это самый жёсткий интеграционный тест, с которым может столкнуться игровой движок на Z80.

21.1 Архитектура проекта

Прежде чем написать хотя бы одну строку Z80-кода, тебе нужна структура каталогов, которая масштабируется. Игра для 128K с пятью уровнями, набором тайлов, листами спрайтов, музыкальной партитурой и звуковыми эффектами

генерирует десятки файлов данных. Если ты не организуешь их с самого начала, ты утонешь.

Структура каталогов

```

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/
    Makefile             -- compiled output (gitignored)
                           -- the build system

```

Каждый исходный файл сосредоточен на одной подсистеме. Каждый файл данных проходит через конвейер преобразования, прежде чем попадёт к ассемблеру. Каталог tools/ содержит Python-скрипты, которые конвертируют удобные для художника форматы (PNG-изображения, карты редактора Tiled) в бинарные данные, готовые для ассемблера.

Система сборки

Makefile – это хребет проекта. Он должен:

1. Конвертировать всю графику из PNG в бинарные данные тайлов/спрайтов
2. Конвертировать карты уровней из формата Tiled в бинарные тайловые карты
3. Сжать данные уровней, графические банки и музыку подходящим упаковщиком
4. Ассемблировать все исходные файлы в один бинарник

5. Сгенерировать итоговый файл .tap с правильным загрузчиком

```

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

```

Ключевая идея – конвейер данных. Художник экспортирует PNG-тайлсет из Aseprite. Скрипт `png2tiles.py` нарезает его на тайлы 8x8 или 16x16, конвертирует каждый в чересстрочный пиксельный формат Spectrum и записывает бинарный блоб. Дизайнер уровней экспортирует карту `.tmx` из Tiled. Скрипт `map2bin.py` извлекает индексы тайлов и записывает компактную бинарную тайловую карту. Упаковщик сжимает каждый блоб. И только тогда ассемблер подключает результат через INCBIN в нужный банк памяти.

Этот конвейер означает, что контент игры всегда в редактируемой форме (PNG, TMX), а система сборки обрабатывает каждое преобразование автоматически. Измени файл в PNG, набери `make`, и новый тайл появится в игре.

21.2 Карта памяти: распределение банков 128K

ZX Spectrum 128K имеет восемь 16-килобайтных банков ОЗУ, пронумерованных от 0 до 7. В любой момент процессор видит 64-килобайтное адресное пространство:

```
$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
```

Банки 5 и 2 жёстко привязаны к адресам \$4000 и \$8000 соответственно. Только верхнее 16-килобайтное окно (\$C000-\$FFFF) переключается. Регистр выбора банка по порту \$7FFD также управляет отображаемым экраном (банк 5 или банк 7) и активной страницей ПЗУ.

```
; 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
```

Критически важное правило: **всегда сохраняй последнюю запись в \$7FFD** в теневой переменной. Порт \$7FFD – только для записи, прочитать текущее состояние нельзя. Если тебе нужно изменить один бит (скажем, переключить экран), не нарушая выбор банка, ты должен прочитать теневую переменную, изменить нужный бит, записать результат и в порт, и в теневую переменную.

Распределение банков Ironclaw

Вот как Ironclaw распределяет свои 128 килобайт по восьми банкам:

<pre>Tileset graphics (compressed) Decompression buffer</pre>	<pre>Bank 1 (\$C000) -- Level data: tilemaps for levels 3-5 (compressed) Boss level data and patterns Enemy spawn tables</pre>
<pre>Bank 2 (\$8000) -- FIXED: Main game code Player logic, physics, collisions Sprite routines, entity system State machine, HUD ~ 14KB code, 2KB tables/buffers</pre>	<pre>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</pre>

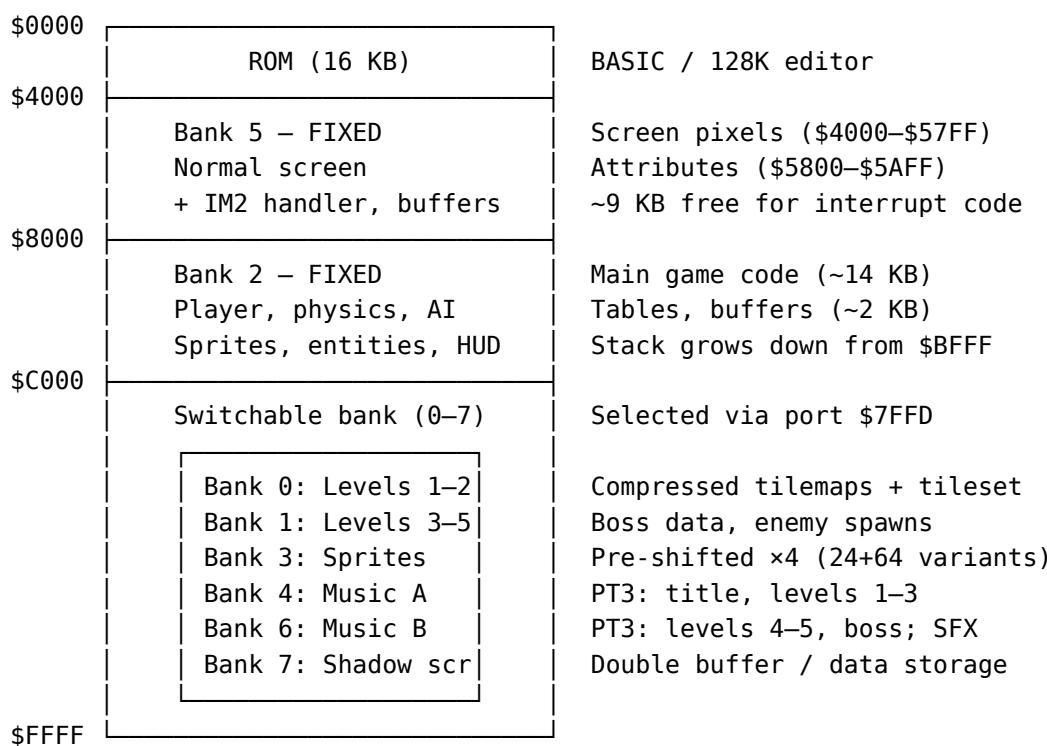
Projectiles, particles, pickups
~ 12KB total

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



Key: Banks 2 and 5 are always visible (hardwired).

Only \$C000–\$FFFF is switchable.

Port \$7FFD is write-only – always shadow its state!

Несколько замечаний об этой раскладке:

Код размещён в банке 2 (фиксированном). Поскольку банк 2 всегда отображён по адресам \$8000–\$BFFF, основной код игры всегда доступен. Тебе никогда не нужно подключать код – только данные. Это исключает самый опасный класс ошибок с банками: вызов подпрограммы, которая была выгружена.

Графика спрайтов в банке 3, отдельно от данных уровней в банках 0-1. При рендеринге кадра рендереру нужна и графика тайлов (для скроллящегося фона), и графика спрайтов (для сущностей). Если бы и то, и другое было в одном переключаемом банке, пришлось бы переключаться туда-сюда в процессе рендеринга. Размещая их в разных банках, можно подключить данные тайлов, отрисовать фон, затем подключить данные спрайтов и отрисовать все сущности – всего два переключения банков за кадр.

Музыка разделена между банками 4 и 6. Проигрыватель PT3 работает в обработчике прерываний IM2, который срабатывает раз за кадр. Обработчик прерываний должен подключить банк с музыкой, обновить регистры AY и переключить обратно на тот банк, который использовал основной цикл. Разделение музыки на два банка означает, что обработчик прерываний должен знать, в каком банке находится текущая композиция. Мы решаем это с помощью переменной:

```
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
```

Теневой экран в банке 7 доступен для двойной буферизации при обновлении скроллинга (как описано в Главе 17). Когда ты не ведёшь активную двойную буферизацию – в меню, между уровнями, во время заставок – банк 7 является 16 килобайтами свободного хранилища. Ironclaw использует его для хранения распакованной тайловой карты текущего уровня во время геймплея, освобождая переключаемые банки для графики и музыки.

Стек

Стек располагается в верхней части адресного пространства банка 2, растёт вниз от \$BFFF. При ~14 килобайтах кода, начинающегося с \$8000, для стека остаётся примерно 2 килобайта – более чем достаточно для нормальной глубины вызовов, но нужно быть бдительным. Глубокая рекурсия – не вариант. Если ты используешь стековый вывод спрайтов (метод PUSH из Главы 16), помни, что ты заимствуешь указатель стека и должен сохранить и восстановить его с отключёнными прерываниями.

21.3 Конечный автомат

Игра – это не одна программа. Это последовательность режимов – титульный экран, меню, геймплей, пауза, конец игры, таблица рекордов – каждый со своей обработкой ввода, своим рендерингом и своей логикой обновления. В Главе 18 мы представили паттерн конечного автомата. Вот как Ironclaw реализует его на верхнем уровне.

```
; 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

```

Каждый обработчик состояния полностью владеет кадром. Обработчик геймплея выполняет ввод, физику, ИИ, рендеринг и обновление интерфейса. Обработчик меню читает ввод и рисует меню. Обработчик паузы просто ждёт клавишу снятия паузы, отображая надпись «PAUSED».

Переходы между состояниями происходят записью нового значения в `current_state`. Переход из `STATE_GAMEPLAY` в `STATE_PAUSE` не требует очистки – игровое состояние не затрагивается, и возврат в `STATE_GAMEPLAY` продолжает ровно с того места, где остановился. Но переход из `STATE_GAMEOVER` в `STATE_HISCORE` требует проверки, попадает ли счёт игрока в таблицу рекордов, а переход из `STATE_LEVELWIN` в `STATE_GAMEPLAY` требует загрузки и распаковки данных следующего уровня.

21.4 Кадр геймплея

Именно здесь происходит интеграция. В состоянии `STATE_GAMEPLAY` каждый кадр должен выполнить следующее, в указанном порядке:

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

На Pentagon с его 71 680 тактами на кадр остаётся 21 000-28 000 тактов запаса. Звучит комфортно, но это обманчиво. Эти оценки – средние значения. Когда на экране четыре врага, а игрок прыгает через пропасть с летящими снарядами, худший случай может быть на 20-30% выше среднего. Твой запас – это запас прочности.

Порядок имеет значение. Ввод должен быть первым – тебе нужно намерение игрока до моделирования физики. Физика должна предшествовать обнаружению столкновений – нужно знать, куда сущности хотят двигаться, прежде чем проверять, могут ли они. Реакция на столкновения должна предшествовать рендерингу – нужны окончательные позиции, прежде чем что-то рисовать. А спрайты должны рисоваться после фона, потому что спрайты накладываются на тайлы.

Чтение ввода

```
; 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

```

Обрати внимание, что чтение клавиатуры использует IN A, (C) с адресом полуяда в В. Каждая клавиша соответствует биту в байте результата. Объединение клавиатуры и джойстика в один байт означает, что остальной логике игры безразлично, какое устройство ввода использует игрок.

Движок скроллинга

Скроллинг – самая дорогая операция в кадре. Глава 17 подробно рассматривала техники; здесь показано, как они интегрируются в игру.

Ironclaw использует метод **комбинированного скроллинга**: скроллинг с гранулярностью символа (скачки по 8 пикселей) для основного видового окна с пиксельным смещением (0-7) внутри текущего 8-пиксельного окна для плавного визуального перемещения. Когда пиксельное смещение достигает 8, видовое окно сдвигается на один столбец тайлов, а смещение сбрасывается в 0.

Видовое окно имеет ширину 30 символов (240 пикселей) и высоту 20 символов (160 пикселей), оставляя место для 2-символьного интерфейса сверху и снизу. Тайловая карта уровня обычно имеет ширину 256-512 тайлов и высоту 20 тайлов.

Когда видовое окно сдвигается на один столбец тайлов, рендерер должен:

1. Скопировать 29 столбцов текущего экрана на один символ влево (или вправо)
2. Нарисовать новый открывшийся столбец тайлов из тайловой карты

Копирование столбцов – это цепочка LDIR: 20 рядов x 8 пиксельных строк x 29 байт = 4 640 байт по 21 такту каждый = 97 440 тактов. Это больше, чем целый кадр. Вот почему техника теневого экрана из Главы 17 критически важна.

```

; 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

```

Но даже с двойной буферизацией полное копирование столбцов обходится дорого. Ironclaw оптимизирует это, распределяя работу: во время плавного субтайлового скроллинга (пиксельное смещение 1-7) копирование столбцов не происходит – меняется только смещение. Дорогое копирование столбцов происходит только на границах тайлов, примерно каждые 4-8 кадров в зависимости от скорости игрока. Между этими пиками рендеринг скроллинга практически бесплатен.

Когда граница тайла пересекается, копирование столбцов можно распределить на два кадра с помощью двойного буфера: кадр N рисует верхнюю половину сдвинутого экрана в задний буфер, кадр N+1 рисует нижнюю половину и переключает. Игрок видит бесшовный скроллинг, потому что переключение происходит только когда задний буфер полностью готов.

21.5 Интеграция спрайтов

Ironclaw использует спрайты OR+AND с маской (Глава 16, метод 2) для всех игровых сущностей. Это стандартная техника: для каждого пикселя спрайта выполняется AND с байтом маски для очистки фона, затем OR с данными спрайта для установки пикселей.

Каждый спрайт 16x16 имеет четыре предварительно сдвинутых копии (Глава 16, метод 3), по одной для каждого 2-пиксельного горизонтального выравнивания. Это превращает попиксельный сдвиг из операции времени выполнения в обращение к таблице подстановки. Цена: каждый кадр спрайта требует 4 варианта x 16 строк x 3 байта/строку (2 байта данных + 1 байт маски, расширенные до 3 байт для обработки переполнения сдвига) = 192 байта. Зато скорость рендеринга падает с ~1 500 тактов до ~1 000 тактов на спрайт, и при 8-10 спрайтах на экране эта экономия накапливается.

Предварительно сдвинутые данные спрайтов хранятся в банке 3. Во время фазы рендеринга спрайтов рендерер подключает банк 3, проходит по всем активным сущностям и рисует каждую:

```

; 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

```

Восстановление фона (грязные прямоугольники)

Перед рисованием спрайтов в новых позициях нужно стереть их из старых позиций. Ironclaw использует метод грязных прямоугольников из Главы 16: перед рисованием спрайта сохраняем фон под ним в буфер. Перед проходом рендеринга спрайтов следующего кадра восстанавливаем эти сохранённые фоны.

```

; 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

```

Стоимость грязных прямоугольников пропорциональна количеству и размеру спрайтов. Для 8 существ размером 16x16 пикселей (3 байта в ширину после сдвига) сохранение и восстановление стоит примерно $8 \times 16 \times 3 \times 2$ (сохранение + восстановление) $\times \sim 10$ тактов/байт = $\sim 7\,680$ тактов. Недёшево, но предсказуемо.

21.6 Столкновения, физика и ИИ в контексте

Главы 18 и 19 рассматривали эти системы изолированно. В интегрированной игре ключевая задача – порядок: какая система запускается первой и какие данные каждой из них нужны от остальных?

Цикл физика-столкновения

Обновление физики должно чередоваться с обнаружением столкновений. Паттерн такой:

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

Горизонтальное и вертикальное перемещения разделены, потому что реакция на столкновение должна обрабатывать каждую ось независимо. Если тыдвигаешься по диагонали и попадаешь в угол, нужно скользить вдоль стены по одной оси, останавливаясь по другой. Одновременная проверка обеих осей приводит к багам «залипания», когда игрок застревает на углах.

Все позиции используют формат с фиксированной точкой 8.8 (Глава 4): старший байт – пиксельная координата, младший – дробная часть. Значения скоростей тоже 8.8. Это даёт субпиксельную точность движения без какого-либо умножения в ядре физического цикла – сложения и сдвигов достаточно.

```
; 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
```

Столкновение с тайлами

Проверка столкновения с тайлом преобразует пиксельную координату в индекс тайла, а затем ищет тип тайла в карте столкновений уровня:

```
; 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

```

В Ironclaw ширина уровней установлена в 256 тайлов. Это не совпадение – это делает умножение на ширину ряда тривиальным (номер ряда становится старшим байтом смещения). Уровень шириной 256 тайлов по 8 пикселей на тайл – это 2 048 пикселей, примерно 8,5 экранов в ширину. Для более длинных уровней можно использовать ширину 512 тайлов (умножение ряда на 2 через SLA E : RL D), хотя это стоит несколько дополнительных тактов на каждое обращение.

ИИ врагов

Каждый тип врага имеет конечный автомат (Глава 19). Состояние хранится в структуре сущности:

```

; 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
```

Четыре типа врагов Ironclaw:

1. **Walker** - Патрулирует между двумя точками. Когда игрок оказывается в пределах 64 пикселей по горизонтали, переключается в состояние преследования (идёт к игроку). Переключается в атаку (контактный урон) при столкновении. Возвращается к патрулированию, когда игрок удаляется или враг достигает края платформы.
2. **Flyer** - Синусоидальное вертикальное движение (с использованием таблицы синусов из Главы 4). Игнорирует столкновения с тайлами. Преследует игрока по горизонтали, когда тот в зоне досягаемости. Сбрасывает снаряды через интервалы.
3. **Shooter** - Стационарный. Стреляет горизонтальным снарядом каждые N кадров, когда игрок находится в прямой видимости (тот же ряд тайлов, между ними нет сплошных тайлов). Снаряд – это отдельная сущность, выделяемая из пула сущностей.
4. **Boss** - Многофазный конечный автомат. Фаза 1: патрулирование платформы, стрельба веером. Фаза 2 (ниже 50% здоровья): ускоренное движение, прицельная стрельба, вызов Walker. Фаза 3 (ниже 25% здоровья): ярость, непрерывный огонь, тряска экрана.

Ключевая оптимизация из Главы 19: ИИ не запускается каждый кадр. Обновления ИИ врагов распределяются по кадрам с помощью простого циклического перебора:

```
; 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

```

Это означает, что ИИ каждого врага запускается раз в 3 кадра. При 50 fps это всё ещё ~17 обновлений ИИ в секунду на врага – более чем достаточно для отзывчивого поведения. Экономия существенна: если ИИ стоит ~500 тактов на врага, запуск всех 8 врагов каждый кадр обходится в 4 000 тактов. Запуск 2-3 врагов за кадр – 1 000-1 500 тактов. Физика и обнаружение столкновений по-прежнему работают каждый кадр для плавного движения.

21.7 Интеграция звука

Музыка

Проигрыватель РТЗ работает внутри обработчика прерываний IM2, как показано в разделе 21.2. Проигрыватель занимает примерно 1,5-2 килобайта кода и выполняется раз за кадр, потребляя ~2 500-3 500 тактов в зависимости от сложности текущего ряда паттерна.

Каждый уровень имеет свой музыкальный трек. При переходе между уровнями игра:

1. Затухает текущий трек (плавное уменьшение громкости AY до 0 за 25 кадров)
2. Подключает нужный банк с музыкой (банк 4 или 6)
3. Инициализирует проигрыватель РТЗ начальным адресом новой композиции
4. Плавно вводит звук

Формат данных РТЗ компактен – типичный 2-3-минутный игровой музыкальный цикл сжимается до 2-4 килобайт с Pletter, поэтому два банка для музыки (4 и 6) вмещают все шесть треков (титульный, пять уровней, босс, конец игры).

Звуковые эффекты

Звуковые эффекты используют систему захвата каналов на основе приоритетов из Главы 11. Когда срабатывает звуковой эффект (прыжок игрока, гибель врага, выстрел снаряда), движок SFX временно захватывает один канал AY, подменяя

то, что музыка делала на этом канале. Когда эффект заканчивается, канал возвращается под управление музыки.

```
; 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
```

Обновление SFX выполняется внутри обработчика прерываний, после проигрывателя PT3. Если SFX активен, он перезаписывает значения регистров AY, которые проигрыватель PT3 только что установил для захваченного канала. Это означает, что музыка продолжает корректно играть на двух других каналах, а захваченный канал воспроизводит звуковой эффект.

Определения SFX – это процедурные таблицы, а не сэмплы. Каждая запись – последовательность покадровых значений регистров:

```
; 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

```

Такой подход потребляет пренебрежимо мало памяти (8-20 байт на эффект) и пренебрежимо мало процессорного времени (несколько десятков тактов за кадр на запись 3-4 значений регистров AY).

21.8 Загрузка: лента и DivMMC

Игра для ZX Spectrum должна как-то загружаться. В 1980-х это означало ленту. Сегодня у большинства пользователей есть DivMMC (или аналог) с SD-картой под управлением esxDOS. Ironclaw поддерживает оба варианта.

Файл .tap и загрузчик на BASIC

Формат файла .tap – это последовательность блоков данных, каждый из которых предваряется 2-байтной длиной и флаговым байтом. Программа-загрузчик на BASIC (сама являющаяся блоком в .tap) использует команды LOAD "" CODE для загрузки каждого блока по нужному адресу.

Структура .tap файла Ironclaw:

```

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)

```

Загрузчик на BASIC:

```

20 LOAD "" SCREEN$
30 LOAD "" CODE
40 BORDER 0: PAPER 0: INK 0: CLS
50 RANDOMIZE USR 32768

```

Строка 10 устанавливает RAMTOP ниже \$8000, защищая наш код от стека BASIC. Стока 20 загружает экран загрузки непосредственно в экранную память (команда LOAD "" SCREEN\$ Spectrum делает это автоматически). Стока 30 загружает основной блок кода. Стока 40 очищает экран. Стока 50 прыгает на наш код по адресу \$8000.

Но это загружает только основной блок кода. Банковые данные (блоки 3-7) должны загружаться нашим собственным Z80-кодом, который подключает каждый банк и использует подпрограмму загрузки с ленты из ПЗУ:

```
; 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
```

Загрузка через esxDOS (DivMMC)

Для пользователей с DivMMC или аналогичным оборудованием загрузка с SD-карты драматически быстрее и надёжнее. API esxDOS предоставляет файловые операции через RST \$08 с последующим номером функции:

```
; 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 определяет наличие esxDOS при запуске, проверяя сигнатуру DivMMC. При наличии загружает все данные из файлов на SD-карте вместо ленты:

```

; 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
```

Код обнаружения:

```
; 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
```

На практике самый надёжный метод обнаружения проверяет идентификационный байт DivMMC по известному адресу ловушки или использует заведомо безопасный вызов RST \$08. Метод выше работает, потому что если esxDOS отсутствует, RST \$08 прыгает на обработчик ошибок ПЗУ, который для 128К ПЗУ по адресу \$0008 представляет собой безобидный возврат с очищенным флагом переноса. В продакшн-коде следует использовать более надёжную проверку; приведённый паттерн иллюстрирует концепцию.

21.9 Экран загрузки, меню и таблица рекордов

Экран загрузки

Экран загрузки – первое впечатление игрока. Он загружается командой LOAD "" SCREEN\$ в BASIC-загрузчике, что означает, что он появляется, пока оставшиеся блоки данных грусятся с ленты. При загрузке через esxDOS она настолько

быстра, что стоит отображать экран минимальное время:

```
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
```

Экран загрузки – это стандартный файл экрана Spectrum: 6 144 байта пиксельных данных, за ними 768 байт атрибутов, итого 6 912 байт. Создай его в любом Spectrum-совместимом графическом редакторе (ZX Paintbrush, SEViewer или Multipaint) или конвертируй современное изображение инструментом дизайна.

Титульный экран и меню

Состояние титульного экрана отображает логотип игры и анимированный фон, затем переходит в меню при любом нажатии клавиши:

```
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
```

Меню предлагает три пункта: Начать игру, Настройки, Таблица рекордов. Навигация – клавишами вверх/вниз, выбор – огнём/Enter. Меню – это простой

конечный автомат внутри обработчика STATE_MENU:

```

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

```

Таблица рекордов

Таблица рекордов хранится в виде 10 записей в области данных банка 2:

```

; 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

```

Очки используют BCD (двоично-десятичный код) – две десятичные цифры на байт, три байта на счёт, что даёт максимум 999 999 очков. BCD предпочтительнее двоичного формата для отображения, потому что преобразование 24-битного двоичного числа в десятичное на Z80 требует дорогостоящего деления. С BCD инструкция DAA автоматически обрабатывает перенос между цифрами, а для печати достаточно маскировать полубайты:

```

; 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

```

Когда игра заканчивается, код просматривает таблицу рекордов, чтобы определить, попадает ли счёт игрока в неё. Если да, игра переходит в STATE_HISCORE для ввода имени (три символа, выбираемых клавишами вверх/вниз/огонь).

На системах с esxDOS таблица рекордов может сохраняться на SD-карту. На системах с лентой рекорды сохраняются только на время текущей сессии.

21.10 Загрузка уровней и распаковка

Когда игрок начинает уровень или завершает его, игра должна:

1. Подключить банк, содержащий данные уровня (банк 0 для уровней 1-2, банк 1 для уровней 3-5)
2. Распаковать тайловую карту в банк 7 (банк теневого экрана, перепрофилированный в буфер данных при переходах между уровнями)
3. Распаковать графику тайлов в буфер в банке 2 или банке 0
4. Инициализировать массив сущностей из таблицы спавна уровня
5. Сбросить видовое окно на начальную позицию уровня
6. Сбросить состояние движка скроллинга

```

; 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

```

Выбор упаковщика здесь важен. Данные уровня загружаются один раз за уровень (во время экрана перехода), поэтому скорость распаковки не критична – мы можем позволить себе ~250 тактов на байт у Exomizer ради лучшей степени сжатия. Но графика тайлов может нуждаться в распаковке во время геймплея (если тайлы хранятся в банках), поэтому предпочтительны ~69 тактов на байт у Pletter.

Как обсуждалось в Главе 14, код распаковщика сам занимает память. ZX0 с ~70 байтами идеален для проектов с дефицитом пространства для кода. Ironclaw включает и распаковщик ZX0 (для данных уровней при загрузке), и распаковщик Pletter (для потоковых данных тайлов во время геймплея).

21.11 Профилирование с DeZog

Ты написал весь код. Он компилируется. Он работает. Игрок ходит, враги патрулируют, тайлы скроллятся, музыка играет. Но бюджет кадра переполняется на уровне 3, где одновременно на экране шесть врагов и три снаряда. Полоска бордюра показывает красную полосу, выходящую за пределы видимой области экрана. Ты теряешь кадры.

Вот тут DeZog зарабатывает своё место в твоём инструментарии.

Что такое DeZog?

DeZog – это расширение VS Code, предоставляющее полноценную среду отладки для Z80-программ. Он подключается к эмуляторам (ZEsarUX, CSpect или собственному встроенному симулятору) и даёт тебе:

- Точки останова (по адресу, условные, логпойнты)
- Пошаговое выполнение (вход, перешагивание, выход)
- Наблюдение за регистрами (все регистры Z80, обновляемые в реальном времени)
- Просмотр памяти (hex-дамп с живым обновлением)
- Представление дизассемблера
- Стек вызовов
- **Счётчик тактов** – инструмент профилирования, который нам нужен

Рабочий процесс профилирования

Полоска бордюра говорит, что ты выходишь за бюджет. DeZog говорит, *где*.

Шаг 1: Изолируй медленный кадр. Установи условную точку останова в начале основного цикла, которая срабатывает только при установленном флаге «переполнение кадра». Добавь код, устанавливающий этот флаг, когда кадр занимает слишком много:

```
; 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:
```

Шаг 2: Измерь стоимость подсистем. Счётчик тактов DeZog позволяет измерить точную стоимость любого участка кода. Помести курсор в начало

`update_enemy_ai`, запиши значение счётчика тактов, перешагни через вызов и запиши новое значение. Разница – это точная стоимость.

Систематический проход профилирования измеряет каждую подсистему:

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%

Это средний случай. Теперь профилируй худший случай – уровень 3, шесть врагов на экране, игрок возле правого края, вызывающий скроллинг:

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%

Всего 9% запаса в худшем случае. Это опасно мало. Ещё один враг или сложный музыкальный паттерн могут вывести за рамки.

Шаг 3: Найди узкое место. Таблица профилирования делает очевидным: скроллинг + рендеринг тайлов потребляют 41% кадра в худшем случае. Рендеринг спрайтов забирает 19%. ИИ врагов – 8%.

Шаг 4: Оптимизируй узкое место. Варианты, примерно в порядке влияния:

- 1. Распредели стоимость скроллинга.** Вместо рендеринга полного нового столбца за один кадр рисуй половину в кадре N и половину в кадре N+1 с помощью двойного буфера (обсуждалось в разделе 21.4). Это снижает пик скроллинга с ~29 000 до ~15 000 тактов за кадр.

2. **Используй скомпилированные спрайты для игрока.** Спрайт игрока всегда на экране и всегда рисуется. Переход от OR+AND с маской (Глава 16, метод 2) к скомпилированным спрайтам (метод 5) экономит ~30% на каждую отрисовку спрайта, но увеличивает расход памяти. Для одной часто рисуемой сущности этот компромисс оправдан.
3. **Уменьши перерисовку спрайтов.** Если два врага перекрываются, ты рисуешь пиксели, которые будут перезаписаны. Сортируй сущности по Y-координате (от дальних к ближним) и пропускай отрисовку полностью закрытых спрайтов. Это помогает в худшем случае, когда сущности группируются.
4. **Подтяни ИИ.** Профилируй run_entity_ai для каждого типа врага. Проверка прямой видимости Shooter (сканирование столбцов тайлов на предмет загораживания) часто оказывается самой дорогой операцией ИИ. Кэшируй результат: перепроверяй прямую видимость каждые 8 кадров вместо каждого 3.

После оптимизации худший случай падает до ~58 000 тактов, оставляя 19% запаса. Это комфортно.

Конфигурация DeZog для Ironclaw

DeZog подключается к эмулятору, поддерживающему его протокол отладки. Для разработки под ZX Spectrum 128K рекомендуется ZEsarUX:

```
{
  "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
      }
    }
  ]
}
```

Параметр history включает обратную отладку – ты можешь шагать назад,

чтобы увидеть, как ты пришёл к ошибке. Это неоценимо для отслеживания глюков столкновений, когда сущность телепортировалась сквозь стену три кадра назад.

21.12 Конвейер данных в деталях

Перенос данных из инструментов художника в игру – зачастую самая недооценённая часть проекта. Конвейер Ironclaw конвертирует четыре вида ассетов:

Тайлсеты (PNG в пиксельный формат Spectrum)

Художник рисует тайлы в Aseprite, Photoshop или любом другом редакторе пиксельной графики в виде индексированного PNG. Тайлы расположены сеткой на одном листе. Скрипт конвертации:

1. Читает PNG, проверяет, что он 1-битный (чёрно-белый) или индексированный с цветами, совместимыми со Spectrum
2. Нарезает на тайлы 8x8 или 16x16
3. Конвертирует каждый тайл в чересстрочный пиксельный формат Spectrum (где строка 0 находится по смещению 0, строка 1 – по смещению 256, а не по смещению 1 – в соответствии с раскладкой экрана)
4. Опционально дедуплицирует идентичные тайлы
5. Записывает бинарный блоб и таблицу символов, отображающую ID тайлов в смещения

Для атрибутов каждый тайл также несёт цветовой байт (чернила (ink) + фон (paper) + яркость (bright)). Скрипт извлекает его из палитры PNG и записывает параллельную таблицу атрибутов.

Листы спрайтов (PNG в предварительно сдвинутые данные спрайтов)

Спрайты следуют аналогичному конвейеру, но с дополнительным шагом: предварительным сдвигом. Скрипт конвертации:

1. Читает PNG-лист спрайтов
2. Нарезает на отдельные кадры
3. Генерирует маску для каждого кадра (любой не-фоновый пиксель даёт 0 в маске, фоновый – 1)
4. Для каждого кадра генерирует 4 горизонтально сдвинутых варианта (смещение 0, 2, 4, 6 пикселей)
5. Каждый сдвинутый вариант расширяется на один байт (2-байтовый спрайт становится 3-байтовым для хранения переноса сдвига)
6. Записывает чередующиеся байты данных и маски для эффективного рендеринга

Карты уровней (Tiled JSON в бинарную тайловую карту)

Уровни разрабатываются в Tiled – бесплатном кроссплатформенном редакторе тайловых карт. Дизайнер размещает тайлы визуально, добавляет слои объектов для точек спавна сущностей и триггеров и экспортит в JSON или TMX.

Скрипт конвертации:

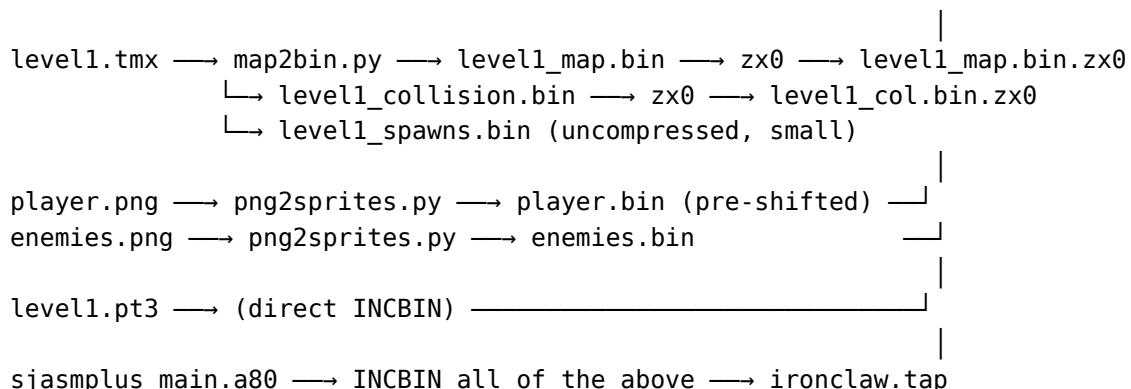
1. Читает экспорт Tiled
2. Извлекает слой тайлов как плоский массив индексов тайлов
3. Извлекает слой объектов для точек спавна (позиции врагов, начальная позиция игрока, расположение предметов)
4. Генерирует карту столкновений: для каждого тайла определяет, является ли он сплошным, платформой, опасностью или пустым (на основе файла свойств тайлов)
5. Записывает тайловую карту, карту столкновений и таблицу спавна как отдельные бинарные файлы

Музыка (Vortex Tracker II в PT3)

Музыка создаётся в Vortex Tracker II, который экспортирует непосредственно в формат .pt3. Файл PT3 встраивается в данные банка через INCBIN. Код проигрывателя PT3 (широко доступный как Z80-ассемблер с открытым исходным кодом, обычно 1,5-2 килобайта) размещается в банке с музыкой рядом с данными композиций.

Сборка воедино

Полный конвейер конвертации для уровня:



Каждый шаг автоматизирован Makefile. Художник меняет тайл, набирает make и видит результат в эмуляторе.

21.13 Формат релиза: сборка .tap

Конечный продукт – файл .tap. sjasmplus может генерировать выходные .tap файлы напрямую через директиву SAVETAP:

```

; 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

```

Точный синтаксис SAVETAP зависит от версии sjasmplus. Для 128К-игр с банковыми данными самый чистый подход – генерировать снапшот .sna (который захватывает состояние всех банков) для тестирования в эмуляторе и .tap с BASIC-загрузчиком плюс блоки машинного кода для дистрибуции.

Тестирование релиза

Перед публикацией протестируй как минимум на трёх эмуляторах:

1. **Fuse** – эталонный эмулятор Spectrum, точный тайминг оригинального оборудования
2. **Unreal Specsy** – тайминг Pentagon, стандарт демосцены, хороший отладчик
3. **ZEsarUX** – поддержка 128К-банков, эмуляция esxDOS, интеграция с DeZog

И если возможно, протестируй на реальном оборудовании с DivMMC. Эмуляторы иногда различаются в граничных случаях тайминга, и игра, которая идеально работает в Fuse, может терять кадры на реальном Spectrum из-за эффектов спорной памяти, которые эмулятор моделирует чуть иначе.

21.14 Финальная полировка

Разница между работающей игрой и готовой игрой – это полировка. Вот чеклист мелких штрихов, которые имеют значение:

Переходы между экранами. Не переключайся между экранами мгновенно. Простое затухание в чёрный (запись уменьшающейся яркости во все атрибуты за 8 кадров) или протирание (очистка столбцов слева направо за 16 кадров) придаёт игре профессиональный вид. Стоимость: пренебрежимая – переходы происходят между игровыми кадрами.

Анимация смерти. Когда игрок погибает, заморозь геймплей на 15 кадров, мигай спрайтом игрока, переключая его чернила (ink) между кадрами, проиграй SFX смерти, затем перерожди. Не телепортируй игрока обратно на чекпойнт просто так.

Тряска экрана. Когда босс приземляется или происходит взрыв, сдвинь видовое окно на 1-2 пикселя на 4-6 кадров. На Spectrum это можно имитировать,

корректируя смещение скроллинга без фактического перемещения тайлов. Это почти бесплатно и добавляет огромную динамику.

Режим привлечения. После 30 секунд на титульном экране без ввода запусти воспроизведение демо – запиши ввод игрока во время тестового прохождения и воспроизведи его. Так аркадные автоматы привлекали прохожих, и для игр на Spectrum это тоже работает.

Циклическая смена цветов. Анимируй текст меню или цвета логотипа, циклически переключая атрибуты через таблицу палитры. 4-байтовый цикл атрибутов обходится практически в ноль процессорного времени и заставляет статичные экраны оживать.

Антидребезг ввода. Игнорируй нажатия клавиш короче 2 кадров. Без антидребезга курсор меню будет проскаакивать мимо пунктов, потому что клавиша удерживалась несколько кадров. Простой счётчик кадров для каждой клавиши решает проблему:

```
; 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)
```

Итого

- **Структура проекта важна.** Разделяй исходные файлы по подсистемам, файлы данных по типам. Используй Makefile для автоматизации полного конвейера от PNG/TMX до .tap.
- **Карта памяти – тщательно.** Код в банке 2 (фиксированный по \$8000), данные уровней в банках 0-1, графика спрайтов в банке 3, музыка в банках 4 и 6, теневой экран в банке 7. Храни теневую копию порта \$7FFD – он только для записи.
- **Обработчик прерываний владеет музыкой.** Обработчик IM2 подключает банк с музыкой, запускает проигрыватель PT3, обновляет SFX и восстанавливает предыдущий банк. Держи его лёгким – максимум ~3 000 тактов.

- **Бюджет кадра геймплея на Pentagon - 71 680 тактов.** Типичный кадр со скроллингом, 8 спрайтами и ИИ стоит ~50 000 тактов в среднем, ~65 000 в худшем случае. Профилируй и оптимизируй худший случай, а не средний.
- **Скроллинг - самая дорогая одиночная операция.** Используй метод комбинированного скроллинга (посимвольный LDIR + пиксельное смещение) с двойной буферизацией через теневой экран. По возможности распределяй копирование столбцов на два кадра.
- **Запускай ИИ врагов каждый 2-й или 3-й кадр.** Физика и обнаружение столкновений работают каждый кадр; решения ИИ можно амортизировать. Это экономит 2 000-3 000 тактов за кадр в худшем случае.
- **Используй esxDOS для современного оборудования.** API RST \$08 / F_OPEN / F_READ / F_CLOSE прост и быстр. Определяй DivMMC при запуске и откатывайся на загрузку с ленты при отсутствии.
- **Профилируй с DeZog.** Полоска бордюра говорит, что ты выходишь за бюджет. DeZog говорит, где. Измеряй каждую подсистему, находи узкое место, оптимизируй его, измеряй снова.
- **Выбирай правильный упаковщик для каждой задачи.** Exomizer или ZX0 для одноразовой загрузки уровней (лучшая степень сжатия). Pletter для потоковой подачи тайлов во время геймплея (быстрая распаковка). Подробный анализ компромиссов – в Главе 14.
- **Полировка не опциональна.** Переходы между экранами, анимации смерти, тряска экрана, антидребезг ввода и режим привлечения – вот что отличает технодемо от игры.
- **Тестируй на нескольких эмуляторах и реальном оборудовании.** Fuse, Unreal Speccy и ZEsarUX моделируют тайминг по-разному. Поведение DivMMC на реальном оборудовании может отличаться от эмулированного esxDOS.

Источники: World of Spectrum (документация по карте памяти ZX Spectrum 128K и порту \$7FFD); Introspec «Data Compression for Modern Z80 Coding» (Hype, 2017); документация API esxDOS (DivIDE/DivMMC wiki); документация расширения DeZog для VS Code (GitHub: maziac/DeZog); документация sjasmplus (директивы SAVETAP, DEVICE, SLOT, PAGE); спецификация формата PT3 Vortex Tracker II; Главы 11, 14, 15, 16, 17, 18, 19 этой книги.

Глава 22: Портирование — Agon Light 2

«Тот же набор инструкций, совершенно другая машина.»

Ты построил игру. Пять уровней, четыре типа врагов, бой с боссом, музыка на AY со звуковыми эффектами, экран загрузки и система меню — всё работает на ZX Spectrum 128K при 3,5 МГц, в 128 килобайтах банковской памяти, рендерясь через ULA, которая не менялась с 1982 года. Каждый байт учтён. Каждый тakt заработан.

Теперь ты собираешься портировать это на машину, у которой тот же набор инструкций процессора, пятикратная тактовая частота, четырёхкратный объём памяти, аппаратные спрайты, аппаратный тайловый скроллинг, SD-карта для загрузки и 24-битное плоское адресное пространство без банков.

Это должно быть просто.

Это не просто. Это *другое* в таких аспектах, которые тебя удивят, и эти сюрпризы научат тебя вещам об обеих машинах, которые ты бы не узнал никаким другим способом.

Тот же ISA, другой мир

Agon Light 2 работает на Zilog eZ80 с частотой 18,432 МГц и 512 КБ плоской оперативной памяти. eZ80 — прямой потомок Z80: он выполняет весь набор инструкций Z80, использует те же имена регистров, те же флаги, те же мемоники. Если ты написал LD A,(HL) на Spectrum и LD A,(HL) на Agon — опкод идентичен. Поведение идентично. Программист Z80 может сесть за Agon и сразу начать писать код.

Но Agon — это не быстрый Spectrum. Это принципиально другая архитектура в знакомой оболочке. Различия делятся на три категории:

Что добавляет eZ80. 24-битные регистры, 24-битную адресацию, адресное пространство в 16 МБ (из которых 512 КБ заполнены), новые инструкции для 24-битной арифметики и систему режимов (ADL против Z80-совместимого), которая управляет шириной регистров и генерацией адресов.

Что заменяет VDP. ULA Spectrum — чип, который считывает видеопамять и рисует экран — заменяется полностью отдельным процессором. VDP Agon — это микроконтроллер ESP32, на котором работает графическая библиотека

FabGL. Он управляет выводом на экран, спрайтами, тайловыми картами и аудио. Процессор eZ80 общается с VDP через высокоскоростной последовательный канал, отправляя командные последовательности. Общей видеопамяти нет. Ты не записываешь пиксели по адресу — ты отправляешь команды сопроцессору.

Что исчезает. Банковая память, спорная память, атрибутная сетка, чересстрочная раскладка экрана, фреймбуфер на 6 912 байт, бордюр как инструмент синхронизации, прямой доступ к фреймбуферу, потактовая синхронизация с лучом. Всё это пропало.

Чтобы портировать нашу игру со Spectrum, нам нужно понять, что переносится напрямую, что требует переписывания, и что нужно продумать заново.

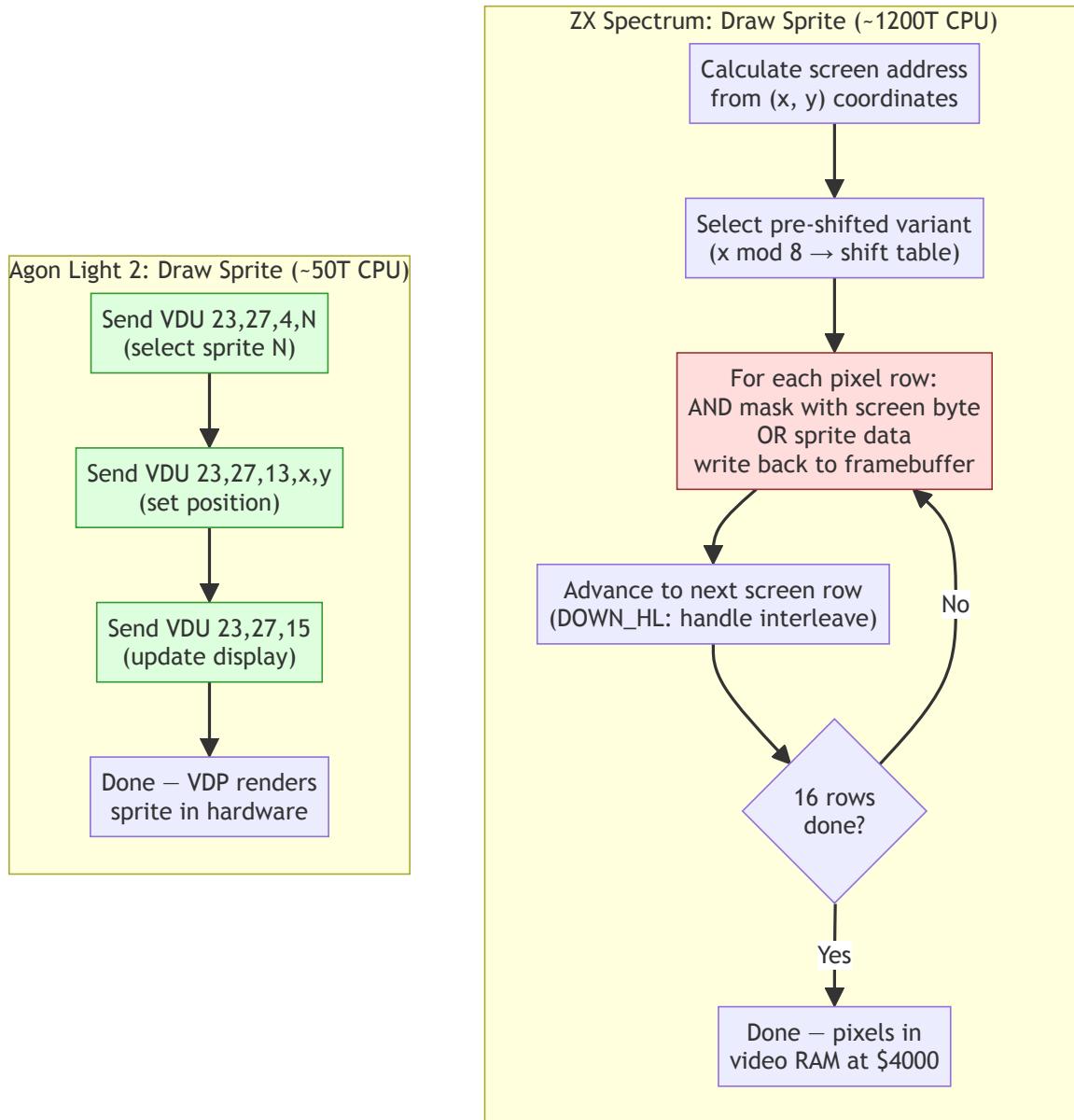
Архитектура с высоты птичьего полёта

Прежде чем погружаться в код, давай разложим две машины бок о бок.

Параметр	ZX Spectrum 128K	Agon Light 2
Процессор	Z80A	eZ80 (Z80-совместимый + расширения ADL)
Тактовая частота	3,5 МГц (7 МГц на турбо-клонах)	18,432 МГц
Оперативная память	128 КБ (8 банков по 16 КБ, переключение через порт \$7FFD)	512 КБ плоская (24-битная адресация)
Адресное пространство	16-битное (64 КБ видно одновременно)	24-битное (16 МБ, 512 КБ заполнены)
Видео	ULA: 256x192, 8x8 атрибутный цвет, прямое отображение в память	VDP (ESP32 + FabGL): множество режимов, до 640x480, спрайты, тайловые карты
Доступ к фреймбуферу	Прямой: запись по адресам \$4000-\$5AFF	Косвенный: отправка команд VDP через последовательный порт
Спрайты	Только программные	Аппаратные: до 256, управляются VDP
Скроллинг	Только программный (сдвиг всего фреймбуфера)	Аппаратный тайловый скроллинг через VDP
Звук	AY-3-8910 (3 канала + шум)	Аудио VDP (синтез ESP32, множество форм волны, ADSR)
Хранилище	Лента / DivMMC (esxDOS)	SD-карта (FAT32, файловый API MOS)
ОС	Нет (голое железо) / esxDOS для файлового ввода-вывода	MOS (Machine Operating System)
Бюджет кадра	~71 680 T-state (Pentagon)	~368 640 T-state (при 50 Гц)

Соотношение бюджетов кадра примерно 5:1. Но это недооценивает реальную

разницу, потому что многие операции, которые потребляют процессорные такты на Spectrum — отрисовка спрайтов, скроллинг экрана, управление фреймбуфером — выгружены на VDP в Agon. Процессор eZ80 тратит свои такты на игровую логику, а не на перестановку пикселей.



Архитектурный сдвиг: На Spectrum процессор сам является движком рендеринга — каждый пиксель размещается инструкциями Z80. На Agon процессор — командный секвенсор — он сообщает VDP, что рисовать, а сопроцессор ESP32 выполняет собственно рендеринг. Затраты процессора падают с ~1 200T до ~50T на спрайт, но теперь ты управляешь асинхронным командным конвейером с последовательной задержкой.

Режим ADL против Z80-совместимого режима

Это единственная самая важная архитектурная концепция для любого программиста Z80, приступающего к eZ80. Сделаешь неправильно — код будет падать так, что отладить сложно. Сделаешь правильно — раскроешь полную мощь чипа.

У eZ80 два рабочих режима:

Z80-совместимый режим (Z80-режим). Регистры 16-битные. Адреса 16-битные. Регистр MBASE предоставляет старшие 8 бит каждого адреса, фактически размещая твоё 64-килобайтное окно где-то в 16-мегабайтном адресном пространстве. Код ведёт себя точно как стандартный Z80 — LD HL,\$4000 загружает 16-битное значение, JP (HL) прыгает по 16-битному адресу (с приклеенным MBASE), PUSH HL кладёт 2 байта в стек.

Режим ADL (Address Data Long). Регистры 24-битные. Адреса 24-битные. LD HL,\$040000 загружает 24-битное значение, JP (HL) прыгает по полному 24-битному адресу, PUSH HL кладёт 3 байта в стек. Это родной режим eZ80.

MOS загружает Agon в режиме ADL. Твоё приложение стартует в режиме ADL. Большая часть программного обеспечения Agon работает целиком в режиме ADL. Но Z80-совместимый режим существует, и понимание взаимодействия между двумя режимами критически важно.

Зачем тебе может понадобиться Z80-режим

Если ты портируешь Z80-код со Spectrum, может возникнуть мысль: «просто переключусь в Z80-режим и запущу свой существующий код». Это работает, до определённого предела. Твои 16-битные адресные вычисления, циклы DJNZ, блоковые копирования LDIR — всё ведёт себя идентично в Z80-режиме. MBASE настроен так, что 16-битные адреса отображаются на правильную область памяти Agon.

Проблема — во взаимодействии со всем остальным. Вызовы API MOS ожидают режим ADL. Команды VDP отправляются через подпрограммы MOS, которые предполагают 24-битные стековые кадры. Если ты в Z80-режиме и вызываешь подпрограмму MOS, стековый кадр будет неправильным — MOS кладёт 3 байта на адрес возврата, твой Z80-код положил 2. Результат — повреждение стека и крэш.

Механизм переключения режимов

eZ80 предоставляет специальные префиксы для переключения режима в рамках одной инструкции:

Префикс	Эффект
.SIS (суффикс)	Выполнить следующую инструкцию в Z80-режиме (короткие регистры, короткие адреса)
.LIS	Выполнить в: длинные регистры, короткие адреса

Префикс	Эффект
.SIL	Выполнить в: короткие регистры, длинные адреса
.LIL	Выполнить в режиме ADL (длинные регистры, длинные адреса)

И для вызовов и переходов:

Инструкция	Из режима	В режим
CALL.IS addr	ADL	Z80
CALL.IL addr	Z80	ADL

Суффикс .IS означает «Instruction Short» — инструкция вызова использует короткие (16-битные) соглашения для адреса возврата. .IL означает «Instruction Long» — вызов кладёт 24-битный адрес возврата.

Вот практический паттерн для вызова MOS из кода в Z80-режиме:

```
; 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 предоставляет RST \$08 как единую точку входа API. Суффикс .LIL чисто обрабатывает переключение режима. После вызова выполнение возвращается к твоему коду в Z80-режиме с правильным состоянием стека.

Практическое правило

Для портирования самый чистый подход: **запускай игровую логику в режиме ADL и переводи свой Z80-код на 24-битные соглашения с самого начала.** Не пытайся работать в Z80-режиме и переключаться туда-сюда при каждом вызове MOS. Накладные расходы на переключение режимов и риск рассогласования стека того не стоят.

Это означает, что твой порт не будет побайтовой копией кода Spectrum. Это будет *перевод*. Алгоритмы те же. Логика та же. Использование регистров в основном то же. Но каждый адрес — 24 бита, каждый PUSH кладёт 3 байта, и каждая загрузка непосредственного адреса несёт лишний байт.

Ловушка MBASE

Если всё же используешь Z80-режим, MBASE определяет старшие 8 бит каждого адреса памяти. При загрузке MOS устанавливает MBASE в \$00, что означает, что Z80-адреса 0000 — —FFFF отображаются на физические адреса \$000000-\$00FFFF. Если твой код или данные находятся выше \$00FFFF (выше

первых 64 КБ), код в Z80-режиме не сможет до них достучаться без изменения MBASE.

Это ловушка для портирующих со Spectrum, которые думают: «У меня 512 КБ, положу данные уровней по адресу \$080000.» В Z80-режиме этого адреса не существует. Ты должен либо использовать режим ADL для доступа к нему, либо установить MBASE в \$08 (чтобы адреса 0000 — FFFF отображались на \$080000-\$08FFFF). Но изменение MBASE влияет на *все* обращения к памяти, включая выборку инструкций — так что твой код тоже должен быть в этом регионе, иначе ты прыгнешь в мусор.

Совет прост: оставайся в режиме ADL. Используй полное 24-битное адресное пространство нативно.

Что переносится напрямую

Меняется не всё. Удивительно большая часть игровой логики Spectrum портируется с минимальными изменениями.

Игровая логика и система сущностей

Система сущностей из Главы 18 — массивы структур, хранящие X, Y, тип, состояние, кадр анимации, скорость, здоровье и флаги — переносится почти дословно. Структура основного цикла (HALT, ввод, обновление, отрисовка, повтор) идентична по концепции, хотя конкретный механизм HALT и прерываний отличается.

Вот цикл обновления сущностей на Spectrum:

```
; 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
```

И на Agon:

```
; 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

```

Логика идентична. Инструкции идентичны. Разница в том, что IX, DE и счётчик команд — все 24-битные. Ассемблер обрабатывает кодирование — LD IX,entities генерирует 24-битный непосредственный операнд вместо 16-битного. Сама структура сущности может быть идентичной, или ты можешь расширить поля позиции до 24-битных для больших карт уровней. Это дизайнерское решение, а не ограничение портирования.

Обнаружение столкновений AABB

Код столкновений из Главы 19 переносится напрямую. Проверки AABB используют 8-битные или 16-битные сравнения — те же инструкции CP, SUB и условных переходов работают идентично на обеих машинах.

```

; 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

```

Арифметика с фиксированной точкой

Все вычисления в формате 8.8 с фиксированной точкой — гравитация, скорость, трение, ускорение — переносятся без изменений. Паттерны сдвига и сложения, 16-битные суммирования, правый сдвиг для трения:

```

; Apply gravity: velocity_y += gravity
; Works identically on both platforms
    ld  a,(ix+ENT_VY_L0)
    add a,GRAVITY_L0
    ld  (ix+ENT_VY_L0),a
    ld  a,(ix+ENT_VY_HI)
    adc a,GRAVITY_HI

```

```
ld    (ix+ENT_VY_HI),a
```

Побайтовая арифметика не зависит от того, номинально 16-битные или 24-битные регистры. Аккумулятор всегда 8-битный. Распространение переноса работает одинаково.

Конечный автомат

Конечный автомат игры (титул, меню, геймплей, пауза, конец игры) использует таблицу переходов, индексированную номером состояния. На Spectrum:

```
; 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
```

На Agon таблица указателей хранит 24-битные адреса:

```
; 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
```

Что изменилось: указатели стали 3 байта вместо 2, поэтому умножение индекса

меняется с *2 на *3, а таблица использует DL (define long) вместо DW (define word). В остальном логика идентична.

Что требует переписывания

Рендеринг: от фреймбуфера к командам VDP

Это самое масштабное изменение в порте. На Spectrum рендеринг означает запись байтов по адресам видеопамяти. Весь конвейер рендеринга — отрисовка спрайтов, очистка экрана, закраска тайлов, скроллинг — это процессорный код, который манипулирует памятью по адресам \$4000-\$5AFF.

На Agon рендеринг означает отправку последовательностей команд VDP. VDP понимает протокол на основе потоков байтов VDU (та же командная система VDU, что используется в BBC BASIC, расширенная командами, специфичными для Agon). Ты отправляешь последовательность байтов в VDP через MOS, и ESP32 их обрабатывает.

Спрайты

На Spectrum (из Главы 16) отрисовка замаскированного спрайта 16x16 обходится примерно в 1 200 T-state процессорного времени — чтение байтов маски, AND с экраном, OR данных спрайта, запись обратно. Ты делаешь это для каждого спрайта, каждый кадр.

На Agon ты загружаешь растровое изображение спрайта *один раз*, а затем перемещаешь его, отправляя обновление позиции:

```
; 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

```

Каждый RST \$10 отправляет один байт в VDP через MOS. Общая процессорная стоимость перемещения спрайта — примерно 13 отправленных байт x ~30 T-state на вызов RST = ~390 T-state. Сравни с ~1 200 T-state на Spectrum для полной отрисовки замаскированного спрайта. А версия для Agon не нуждается в сохранении/восстановлении фона — VDP автоматически компонует спрайты поверх фона.

Компромисс: задержка. VDP обрабатывает команды асинхронно. Между отправкой команды «переместить спрайт» и фактическим появлением спрайта на новой позиции существует задержка последовательной передачи и задержка обработки VDP. Для плавной анимации тебе нужно отправлять все обновления спрайтов в начале кадра и рассчитывать на то, что VDP обработает их до следующего обновления экрана.

Тайловый скроллинг

На Spectrum горизонтальный скроллинг означает сдвиг каждого байта видеопамяти влево или вправо — цепочка инструкций RLC или RRC по сотням байт, потребляющая существенную долю бюджета кадра (мы рассчитывали стоимость в Главе 17). Вертикальный скроллинг требует копирования строк развёртки с учётом чересстрочной раскладки памяти.

На Agon VDP поддерживает аппаратные тайловые карты:

```

; Agon: Set up a tilemap (done once)
; VDU 23, 27, 20, tileSize, 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

```

Восемь байт отправлено. Примерно 240 T-state процессорного времени. На Spectrum полноэкранный горизонтальный пиксельный скроллинг обходится в десятки тысяч T-state. На Agon это выполняется аппаратно практически бесплатно.

Но сначала нужно настроить тайловую карту: загрузить определения тайлов, задать размеры карты, заполнить карту индексами тайлов. Это одноразовая стоимость при загрузке уровня, а не покадровая. На Spectrum твои данные тайлов живут в банковой памяти и рисуются во фреймбуфер твоим собственным кодом. На Agon данные тайлов живут в памяти VDP и рисуются ESP32. Твоя роль меняется с «программиста графического движка» на «секвенсора команд VDP».

Раскладка экрана

Весь кошмар чересстрочной раскладки экрана Spectrum — разделённая адресация, подпрограммы DOWN_HL, тщательные вычисления для преобразования координат (x, y) в адреса памяти — исчезает. VDP Agon работает в экранных координатах. Ты говоришь «нарисуй в точке (100, 50)», и VDP делает всё остальное.

Это означает, что подпрограмма DOWN_HL из Главы 2, таблицы подстановки экранных адресов, вычисления адресов атрибутов — ничего из этого не портируется. Оно просто удаляется. Эквивалентная операция на Agon — «отправить пару координат в VDP».

Что нужно продумать заново

Некоторые паттерны Spectrum настолько глубоко встроены в архитектуру игры, что ты не можешь просто переписать слой рендеринга. Нужно менять сам дизайн.

Архитектура памяти

На Spectrum ты тщательно планировал, какие данные в каком банке:

- Банки 0–3: данные уровней, тайлсеты, графика спрайтов
- Банки 4–6: музыкальные паттерны, звуковые эффекты, таблицы подстановки
- Банк 7: теневой экран для двойной буферизации

Каждое переключение банков стоит записи в порт и ограничивает, какой код может видеть какие данные. Архитектура игры формируется 16-килобайтным окном в 128-килобайтное пространство.

На Agon все 512 КБ видны одновременно. Банков нет. Трюка с теневым экраном нет (VDP обрабатывает двойную буферизацию самостоятельно). Ты можешь держать всю свою игру — все пять уровней, все тайлсеты, все спрайты, всю музыку — в памяти одновременно. Переходы между уровнями не требуют загрузки с ленты или диска; ты просто указываешь на другую область оперативной памяти.

Это упрощает разработку, но также убирает ограничение, которое заставляло делать хорошую архитектуру. На Spectrum ты был вынужден думать о локальности данных, о том, что должно быть совместно резидентным, о последовательностях загрузки. На Agon можно быть небрежным. Не будь небрежным. У Agon 512 КБ, а не бесконечность. Хорошо организованная карта памяти — по-прежнему достоинство.

Типичная разметка памяти Agon для портированной игры:

```
$040000 - $04FFFF Game code (~64 KB)
$050000 - $06FFFF Level data, all 5 levels (~128 KB)
$070000 - $07FFFF Music and SFX data (~64 KB)
$080000 - $0FFFFFF Free / working buffers
```

Всё адресуемо одной инструкцией LD HL,\$070000 — никакого переключения банков, никаких записей в порты.

Загрузка

На Spectrum загрузка с ленты — процесс длиной в минуты с характерным звуковым сопровождением. Даже с DivMMC и esxDOS доступ к файлам — это последовательность вызовов RST \$08:

```
; 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

```

На Agon MOS предоставляет файловый API, который читает напрямую с SD-карты:

```

; 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

```

Паттерн похож — открыть, прочитать, закрыть — но Agon читает с FAT32 на SD-карте, что достаточно быстро, чтобы загружать данные уровней между сценами без видимой задержки. Не нужен экран загрузки. Не нужны подпрограммы загрузки с ленты. Не нужна оптимизация блочной загрузки.

Звук

AY-3-8910 Spectrum программируется прямой записью в аппаратные регистры через порты ввода-вывода. Каждая нота, каждое изменение огибающей, каждый шумовой всплеск — это конкретное значение регистра, записанное в конкретный момент.

Аудио Agon обрабатывается VDP. Ты отправляешь звуковые команды через тот же последовательный канал, что используется для графики:

```

; 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

```

Звуковая система Agon поддерживает множество форм волны (синус, квадрат, треугольник, пила, шум) и ADSR-огибающие для каждого канала — функции, которых нет у AY. Но модель программирования совершенно другая. Ты не можешь писать сырье значения регистров; ты отправляешь высокоуровневые нотные команды. Твой проигрыватель музыки для AY — обработчик прерываний IM2, который читает данные паттернов и обновляет 14 регистров AY каждый кадр — вообще не портируется. Тебе нужен новый музыкальный драйвер, который транслирует данные паттернов в звуковые команды VDP.

Один из подходов: написать тонкий слой абстракции, общий для обеих платформ.

```

; 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

```

Та же сигнатура вызова. Различные внутренности. Игровой код вызывает `sound_play_note`, не зная, на какой платформе он работает.

Ввод

Spectrum считывает состояние клавиатуры опросом порта \$FE с адресами полурядов в аккумуляторе. Джойстик Kempston читается из порта \$1F. Это прямые чтения портов ввода-вывода с конкретными битовыми масками.

Agon читает состояние клавиатуры через MOS:

```

; Spectrum: read keyboard
ld  a,$FD          ; half-row: Q W E R T
in  a,($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

```

Spectrum даёт тебе битовую маску одновременно нажатых клавиш, идеальную для игрового ввода (можно определить несколько нажатых клавиш сразу). API клавиатуры MOS Agon событийный: он выдаёт последнюю нажатую клавишу. Для игрового ввода с одновременным обнаружением клавиш обычно используется карта клавиатуры MOS — область памяти, обновляемая MOS, которая отражает состояние всех клавиш:

```
; 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:
```

Это функционально эквивалентно подходу с битовыми масками Spectrum, просто организовано иначе. Портирование прямолинейно: замени чтения портов чтениями из карты клавиатуры в памяти.

eZ80 на 18 МГц: что по-прежнему важно, а что нет

eZ80 примерно в пять раз быстрее Z80A по частоте тактового генератора. Но многие инструкции eZ80 также выполняются за меньшее число тактов, чем их эквиваленты на Z80 — однобайтовые регистровые инструкции часто завершаются за 1 такт вместо 4. Эффективное ускорение для типичного кода находится где-то между 5x и 20x в зависимости от состава инструкций.

Это кардинально меняет расчёт оптимизации.

Что по-прежнему важно: эффективность внутренних циклов

Даже при 18 МГц и 368 000 T-state на кадр, внутренний цикл по-прежнему имеет значение для ресурсоёмких операций. Если ты выполняешь проверки столкновений с тайловой картой, перебираешь 200 сущностей или обрабатывашь конечные автоматы ИИ для десятков врагов, стоимость каждой итерации горячего цикла накапливается.

Базовые техники оптимизации Z80 — хранение значений в регистрах вместо памяти, использование INC L вместо INC HL где возможно, избегание IX/IY-индексированных инструкций на горячих путях (они несут 2-тактовый штраф за префикс на eZ80, как и 4-тактовый штраф на Z80) — по-прежнему дают измеримые улучшения.

```
; 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

```

Этот паттерн — минимальные обращения к памяти, плотное использование регистров, управление циклом через DJNZ — оптимален и на Z80 при 3,5 МГц, и на eZ80 при 18 МГц. Хороший код — это хороший код.

Что становится неактуальным: трюки экономии памяти

На Spectrum давление памяти определяет большую часть инженерии. Предсдвинутые спрайты хранят 4 или 8 копий каждого спрайта, потребляя 4x-8x памяти, как компромисс скорости и памяти. Сжатие обязательно для размещения данных демо в 128 КБ. Таблицы подстановки тщательно дозированы для баланса точности и расхода памяти. Переключение банков добавляет архитектурную сложность с единственной целью адресации более 64 КБ за раз.

На Agon, с 512 КБ плоской оперативной памяти и SD-картой для всего, что не обязательно держать в памяти, эти техники экономии не нужны. Ты можешь хранить 8 предсдвинутых копий каждого спрайта, не беспокоясь о памяти. Можешь держать все таблицы подстановки с полной точностью. Можешь держать все пять уровней в памяти одновременно.

Это не значит, что нужно расточительно расходовать память. Но решения по оптимизации, продиктованные дефицитом памяти — «использовать таблицу синусов на 256 байт или на 128?» — становятся неактуальными. Используй 256. Используй 1 024, если точность помогает.

Что становится неактуальным: самомодифицирующийся код (SMC)

На Spectrum самомодифицирующийся код (SMC) — стандартная техника оптимизации. Ты пишешь инструкции, которые патчят собственные непосредственные операнды, чтобы избежать обращений к памяти:

```

; Spectrum: self-modifying code for speed
ld a,0          ; operand patched at runtime
; ... (the $00 after LD A is overwritten with the real value)

```

На eZ80 SMC по-прежнему работает (у eZ80 нет кэша инструкций, который бы инвалидировался), но мотивация слабее. Дополнительная стоимость обращения к памяти меньше относительно общего бюджета кадра. Важнее то, что MOS отображает некоторые области памяти как доступные только для чтения, и определённые версии прошивки Agon могут ограничивать выполнение модифицированного кода в зависимости от области памяти. SMC не запрещён на Agon, но он редко необходим и может вызывать трудноуловимые проблемы.

Что становится неактуальным: стековые трюки для рендеринга

На Spectrum злоупотребление указателем стека для быстрого заполнения экрана (DI, установить SP на адрес экрана, многократный PUSH данных) — классический приём, потому что PUSH записывает 2 байта за 11 T-state — быстрее любого другого механизма записи. На Agon ты вообще не пишешь во фреймбуфер. VDP обрабатывает рендеринг. Стековые трюки для вывода на экран бессмысленны.

Сравнительная таблица

Вот сравнение одной и той же игры на обеих платформах. Эти числа характерны для платформера, построенного в Главах 21–22.

Метрика	ZX Spectrum 128K	Agon Light 2
Размер игрового кода	~12 КБ	~14 КБ
Код рендеринга	~5 КБ (спрайтовый движок, скроллинг, управление экраном)	~2 КБ (командные последовательности VDP)
Итого кода	~17 КБ	~16 КБ
Данные уровней (все 5)	~40 КБ (сжатые, загружаются поуровнево)	~60 КБ (несжатые, все резидентны)
Графика спрайтов/тайлов	~20 КБ (упакованные, 1bpp + маски)	~80 КБ (8bpp RGBA, загружены в VDP)
Музыка + SFX	~16 КБ (PT3 + таблицы SFX)	~20 КБ (конвертированный формат + данные форм волны)
Итого данных	~76 КБ (помещается в 128 КБ с банками)	~160 КБ (легко помещается в 512 КБ)
Нужно сжатие?	Да, обязательно	Нет, опционально
Стоимость отрисовки спрайта	~1 200 Т/спрайт (программная)	~400 Т/спрайт (команды VDP)
Стоимость скроллинга за кадр	~15 000–30 000 Т (программный сдвиг)	~240 Т (команда смещения VDP)
Бюджет кадра	~71 680 Т	~368 640 Т
Достижимый fps	25–50 (зависит от числа сущностей)	60 (ограничено VDP, не процессором)
Сложность разработки	Высокая (банковая память, раскладка экрана, движок рендеринга)	Средняя (протокол VDP, API MOS, режим ADL)

Метрика	ZX Spectrum 128K	Agon Light 2
Визуальный результат	Монохромные или 2-цветные спрайты в ячейке, атрибутный цвет, 256x192	Полноцветные спрайты, плавный скроллинг, 320x240 или выше

Разница в размере кода показательна. На Spectrum движок рендеринга составляет существенную долю кодовой базы. На Agon команды VDP заменяют большую часть этого кода. Но игра на Spectrum меньше по общему объёму данных, потому что всё сжато и плотно упаковано. Версия для Agon использует больше памяти для более богатых ресурсов (спрайты по 8 бит на пиксель вместо 1 бита на пиксель с масками).

Сравнение бюджетов кадра — самое впечатляющее число. Игра на Agon *простаивает большую часть кадра*. После обработки игровой логики, отправки обновлений спрайтов и обработки ввода, eZ80 нечего делать до следующего кадра. На Spectrum ты борешься за каждый такт, чтобы всё уместить в бюджет.

Процесс портирования: пошагово

Вот практическая последовательность для портирования игры из Главы 21 на Agon.

Шаг 1: Настрой проект Agon

Создай новый проект с инструментарием Agon. Тебе понадобятся:

- Ассемблер eZ80 (ez80asm или инструменты Zilog ZDS)
- Заголовочный файл API MOS (mos_api.inc), определяющий номера функций и константы
- Способ передать бинарник на Agon (SD-карта или загрузка по последовательному порту)

Точка входа отличается от Spectrum. Вместо ORG \$8000 с голым JP на стартовый адрес, Agon загружает исполняемые файлы по адресу \$040000, и MOS передаёт управление по этому адресу:

```
; 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
```

Шаг 2: Замени слой рендеринга

Это основной объём работы. Удали код рендеринга Spectrum и замени его командными последовательностями VDP:

1. **Загрузи растровые изображения спрайтов в VDP.** Конвертируй свои 1bpp-данные спрайтов в формат VDP (обычно 8bpp RGBA). Отправь пиксельные данные командами определения спрайтов VDU 23,27.
2. **Загрузи тайлсет в VDP.** Конвертируй свои 8x8-тайлы из формата атрибутов Spectrum в формат тайлов VDP. Определи размеры тайловской карты и заполни её индексами тайлов для текущего уровня.
3. **Замени подпрограмму отрисовки спрайтов** командами позиционирования спрайтов VDU 23,27 (как показано ранее).
4. **Замени подпрограмму скроллинга** командами смещения тайловой карты VDU 23,27.
5. **Удали код вычисления адресов экрана**, подпрограмму DOWN_HL, подпрограммы адресов атрибутов и все прямые записи во фреймбуфер.

Шаг 3: Транслируй игровую логику

Пройди по коду игровой логики (обновление сущностей, обнаружение столкновений, физика, ИИ, конечный автомат) и адаптируй под режим ADL:

- Замени все DW (define word) на DL (define long) для таблиц адресов.
- Измени указательную арифметику с 16-битной на 24-битную там, где задействованы адреса.
- Проверь, что пары PUSH/POP сбалансированы — каждый push теперь кладёт 3 байта, а не 2.
- Убедись, что счётчики блокового копирования LDIR и LDDR корректны (BC в режиме ADL 24-битный; если твой счётчик помещается в 16 бит, старший байт должен быть нулём).

Шаг 4: Перепиши звук

Напиши новый музыкальный драйвер, который читает данные паттернов и генерирует звуковые команды VDP вместо записей в регистры АУ. Формат данных паттернов может остаться прежним; меняется только выходная стадия.

Шаг 5: Перепиши ввод

Замени чтения портов чтениями из карты клавиатуры MOS. Подход с картой клавиатуры прост и обеспечивает одновременное обнаружение нажатий.

Шаг 6: Перепиши загрузку

Замени файловые операции esxDOS вызовами файлового API MOS. Паттерн похож; различаются только номера функций и соглашение о вызовах.

Шаг 7: Тестируй и настраивай

Запусти игру. Проверь позиции спрайтов, границы столкновений, скорость скроллинга, синхронизацию звука. Асинхронная обработка VDP означает, что визуальные обновления могут приходить на один кадр позже, чем ожидалось — скорректируй тайминги игры при необходимости.

Чему каждая платформа заставляет тебя учиться

Spectrum учит эффективности на уровне тактов и творческому решению задач в условиях ограничений: ты учишься считать такты, использовать раскладку памяти и изобретать техники вроде мультиколора и эффектов только на атрибутах, которые существуют исключительно благодаря ограничениям железа. Agon учит системной архитектуре: управлению асинхронным сопроцессором, структурированию командных конвейеров и созданию инструментов конвертации ассетов для больших объемов данных. Spectrum делает тебя лучшим оптимизатором; Agon делает тебя лучшим системным проектировщиком. Оба навыка переносимы.

Заметка об инструкциях eZ80

eZ80 добавляет несколько инструкций, которые программист Z80 оценит. Наиболее полезные для разработки игр:

LEA (Load Effective Address). Вычисляет адрес из базового регистра плюс 8-битное знаковое смещение, не модифицируя базовый регистр:

```
; eZ80: LEA IX, IY + offset
; Compute IX = IY + displacement without changing IY
    LEA IX,IY+ENT_SIZE      ; IX points to next entity
```

На Z80 для этого нужны PUSH IY / POP IX / LD DE, ENT_SIZE / ADD IX, DE — четыре инструкции и 40+ T-state. LEA делает это за одну инструкцию.

TST (Test Immediate). Выполняет AND аккумулятора с непосредственным значением и устанавливает флаги, не модифицируя A:

```
; 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
```

На Z80 тебе понадобится BIT 7,A (что не работает с произвольными масками) или PUSH AF / AND mask / POP AF (дорого).

MLT (Multiply). Беззнаковое умножение 8x8, результат в 16-битной регистровой паре:

```
; eZ80: MLT BC
; B * C -> BC (16-bit result)
    ld b,sprite_width
    ld c,frame_number
    mlt bc           ; BC = B * C
```

На Z80 для умножения нужен цикл или таблица подстановки. MLT — одна инструкция. Для игровой логики — вычисление смещений спрайтов, индексов тайловой карты, позиций кадров анимации — это существенное упрощение.

Итого

- Agon Light 2 выполняет тот же набор инструкций Z80, что и Spectrum, но с 24-битной адресацией (режим ADL), 512 КБ плоской оперативной памяти, аппаратными спрайтами и тайловыми картами через сопроцессор VDP и бюджетом кадра в ~5 раз больше.
- **Режим ADL** — родной режим. Запускай свою игру в режиме ADL. Избегай Z80-совместимого режима для чего-либо кроме запуска устаревшего кода, который нельзя преобразовать. Переключение режимов через суффиксы .LIL/.SIS доступно, но добавляет сложность и риск.
- **Игровая логика переносится напрямую:** системы сущностей, обнаружение столкновений, физика с фиксированной точкой, конечные автоматы и ИИ — всё переносится с минимальными изменениями (в основном расширение указателей с 16-битных до 24-битных).
- **Рендеринг нужно переписать:** прямой доступ к фреймбуферу Spectrum заменяется командными последовательностями VDP для спрайтов, тайловых карт и скроллинга. Процессорная стоимость рендеринга падает драматически, но теперь ты управляешь асинхронным командным конвейером.
- **Звук нужно переписать:** записи в регистры AY заменяются звуковыми командами VDP. Данные паттернов могут остаться прежними; меняется только выходной драйвер.
- **Архитектура памяти упрощается:** нет банков, нет трюков с теневым экраном, нет сжатия, навязанного дефицитом. Все ресурсы могут быть резидентными одновременно.
- **Трюки Spectrum, которые становятся неактуальными на Agon:** самомодифицирующийся код (SMC) для скорости, рендеринг через указатель стека, предсдвинутые копии спрайтов для компромисса память/скорость, вычисления адресов черезстрочного экрана, визуальные эффекты на атрибутах.
- **Трюки Spectrum, которые по-прежнему актуальны на Agon:** плотные внутренние циклы, эффективный по регистрам код, ориентированная на данные раскладка структур, предвычисленные таблицы подстановки.
- **Каждая платформа учит разным навыкам:** Spectrum учит потактовой эффективности и творческому решению задач в условиях ограничений; Agon учит системной архитектуре, взаимодействию с сопроцессором и управлению пайплайном данных.
- **eZ80 добавляет полезные инструкции:** LEA для вычисления адресов, MLT для аппаратного умножения, TST для неразрушающего тестирования битов — всё это упрощения, устраняющие многоинструкционные паттерны Z80.

Источники: 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); Главы 15-21 этой книги.

Глава 23: Z80-разработка с помощью ИИ

«Z80 они по-прежнему не знают.» – Introspec (spke), Life on Mars, 2024

Эта книга была частично написана с помощью ИИ. Глава, которую ты сейчас читаешь, была набросана Claude Code. Ассемблер, используемый для сборки примеров — mza от MinZ — был создан с помощью ИИ. Демо-проект «Antique Toy», который документирует эта книга, был написан в цикле обратной связи между человеком и ИИ-агентом. Если тебе от этого некомфортно — хорошо. Этот дискомфорт стоит исследовать.

Это самая самосознательная глава книги. Мы честно рассмотрим, что означает помощь ИИ для Z80-разработки в 2026 году — где она реально помогает, где уверенно ошибается, и где ответ — раздражающее «зависит от ситуации». Мы будем делать это с реальными примерами, реальным кодом и реальными случаями неудач, потому что демосцена никогда не терпела хайпа.

23.1 Историческая параллель: HiSoft C на ZX Spectrum

Прежде чем говорить об ИИ, давай поговорим о другой попытке принести инструменты более высокого уровня на ZX Spectrum.

В 1998 году *Spectrum Expert #02* — тот самый выпуск, где Dark и STS опубликовали свой метод средней точки для 3D (Глава 5) — опубликовал обзор компилятора HiSoft C для ZX Spectrum. Вердикт был неоднозначным. Компилятор производил код, который работал «в 10–15 раз быстрее BASIC». Он поддерживал 33 зарезервированных ключевых слова, предлагал stdio.lib с графикой на уровне возможностей BASIC и включал gam128.h для доступа к банкам памяти 128K.

Но у него не было поддержки чисел с плавающей точкой.

Подумай об этом на секунду. Компилятор С. На машине, где плавающая точка уже обрабатывается ROM-калькулятором RST \$28, сидящим в 16 КБ бесплатного кода. А компилятор не мог его использовать.

Вывод рецензента из *Spectrum Expert* был точен: «полезен для критичного к скорости кода, где float не нужен». Инструмент с чёткими сильными сторонами и жёсткими ограничениями, оценённый честно.

HiSoft Pascal HP4D рассказывал похожую историю. Компилятор занимал 12 КБ, оставляя примерно 21 КБ для программ. Он поддерживал вещественные типы и тригонометрические функции — SIN, COS, SQRT — и «подходил для обработки данных и вычислительной математики». Но 21 КБ для твоей программы на машине, где один несжатый экран занимает 6 912 байт, означает, что ты пишешь маленькие программы или ничего.

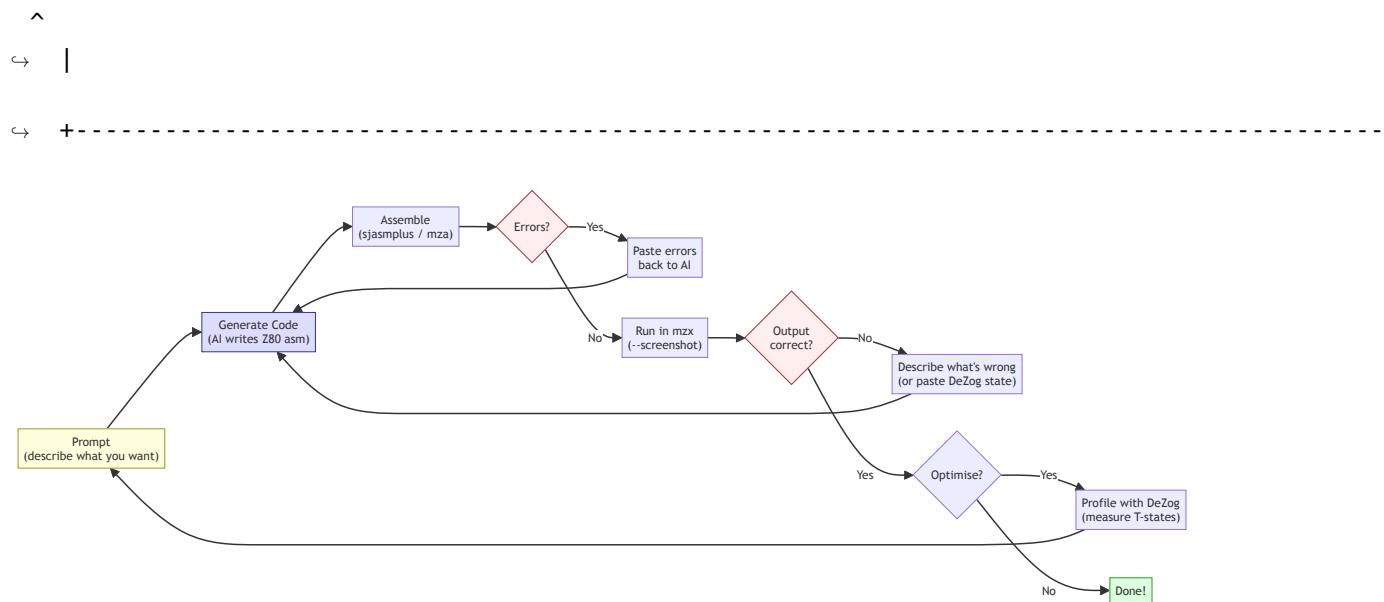
Языки более высокого уровня на ограниченном железе всегда были компромиссом. Они колоссально ускоряют определённые задачи. Делают другие задачи невозможными. Вопрос никогда не был «хорош ли HiSoft C или плох?», а «для чего именно он хорош, и что по-прежнему нужно писать на ассемблере?»

Z80-разработка с помощью ИИ — такой же компромисс. Другой формы, тот же вопрос.

23.2 Цикл обратной связи Claude Code

Вот как Z80-разработка с помощью ИИ работает на практике. Это не магия. Это цикл.

Цикл



Ты описываешь, что хочешь. ИИ генерирует Z80-ассемблер. Ты собираешь его. Сборка падает — неправильный синтаксис, неправильный формат меток, неправильный диалект ассемблера. Ты вставляешь ошибку обратно. ИИ исправляет. Ты собираешь снова. На этот раз компилируется. Запускаешь в эмуляторе. Вывод неправильный — экран заполняется мусором, бордюр не того цвета, ничего не появляется вообще. Описываешь, что видишь. ИИ корректирует. Запускаешь снова.

Этот цикл принципиально не отличается от того, как работает программист-человек, за одним критическим исключением: ИИ генерирует свою первую

попытку гораздо быстрее и допускает определённые классы ошибок гораздо более предсказуемо.

Конкретный пример

Допустим, ты просишь Claude Code написать подпрограмму, которая заполняет область атрибутов ZX Spectrum диагональным полосатым узором. Вот правдоподобная первая попытка:

```
; 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
```

Это разумный код. Он собирается. Работает. Но есть проблемы. Счётчик столбцов C считает вниз от 32, а не вверх от 0, поэтому диагональ идёт не в ту сторону. `or $40` устанавливает бит 6 (BRIGHT), что правильно, но визуальный эффект даёт диагонали справа-сверху вниз-влево, вместо более естественного слева-сверху вниз-вправо. А PUSH/POP AF во внутреннем цикле стоит 22 T-state на каждый байт атрибутов — ненужный расход, если перестроить вычисление.

Ни одна из этих проблем не *ошибка* в том смысле, что код падает. Они *ошибки* в том смысле, что Z80-программист не написал бы код таким образом. Человек, который заполнял атрибуты сотни раз, вычислил бы индекс диагонали иначе, избежал бы PUSH/POP и попал в правильное направление с первой попытки, потому что паттерн `row + column` — это вторая натура.

Вот версия, к которой ты приходишь после двух итераций:

```
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

```

Чище. Без PUSH/POP. Диагонали идут в правильном направлении. Внутренний цикл стоит $4 + 4 + 7 + 4 + 7 + 7 + 6 + 4 + 13 = 56$ T-state на байт — не блестяще, но функционально для подпрограммы заполнения, которая выполняется один раз.

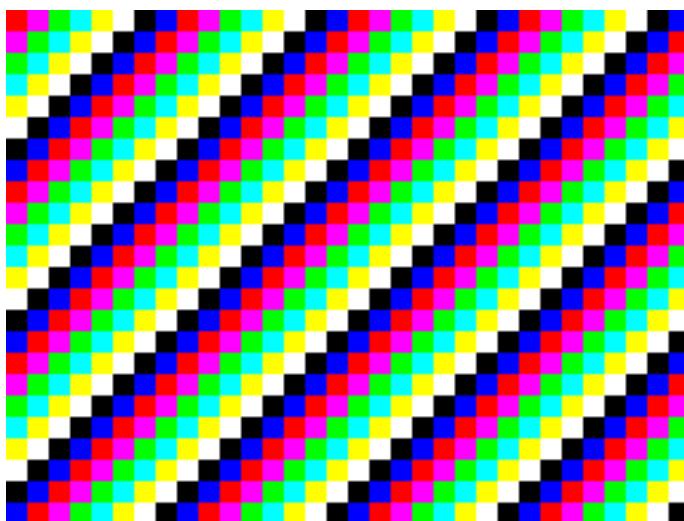


Рис. 41: Сгенерированный ИИ диагональный паттерн заливки с яркими цветными полосами по сетке атрибутов ZX Spectrum

Суть не в том, что ИИ написал плохой код. Суть в том, что *цикл* — запрос, генерация, сборка, тест, исправление, повторный тест — это и есть реальный рабочий процесс. Помощь ИИ не устраняет необходимость понимать Z80. Она смещает узкое место с написания кода на оценку кода.

Что делает цикл быстрым

Цикл быстрее с ИИ, чем без него, для определённых категорий работы:

Шаблонный код. Директива ORG, цикл с HALT, тестовая связка с цветом бордюра из Главы 1, скелет заполнения атрибутов, подпрограмма записи в регистры AY, настройка LDIR, настройка режима прерываний. Каждый Z80-проект начинается с одних и тех же 30-50 строк. ИИ генерирует их правильно и мгновенно. Человек набирает по памяти. ИИ быстрее.

Итерация по известному паттерну. «Теперь сделай диагональ в другую сторону.» «Добавь счётчик кадров, чтобы это анимировалось.» «Сделай так, чтобы цвета циклически менялись через BRIGHT и не-BRIGHT.» Каждая итерация — маленькая дельта к существующему коду. ИИ применяет дельту быстрее, чем ручное редактирование, и изменения обычно корректны.

Генерация тестовых обвязок. «Напиши тест, который заполняет память по адресу \$C000 значениями 0-255, вызывает подпрограмму умножения по адресу \$8000 и проверяет результаты по таблице.» ИИ генерирует такой каркасный код быстро и надёжно. Структура теста — подготовить входные данные, вызвать подпрограмму, сравнить выходные — вполне в компетенции ИИ.

Документация и комментарии. «Добавь подсчёт тактов к каждой инструкции в этом внутреннем цикле.» ИИ знает таблицы таймингов Z80 и применяет их корректно в простых случаях. Это утомительная человеческая работа, с которой машины справляются хорошо.

Что делает цикл медленным

Нестандартные алгоритмы. Когда ты просишь что-то, чего ИИ не видел — новую стратегию развёртки цикла, трюк, использующий поведение флагов Z80 определённым образом, схему генерации кода, заточенную под твою конкретную раскладку данных — ИИ генерирует правдоподобно выглядящий код, который часто тонко ошибается. Хуже того — ошибается так, что код компилируется и работает, но выдаёт неправильные результаты. Ты тратишь больше времени на отладку сгенерированного ИИ кода, чем потратил бы на написание его самостоятельно.

Подсчёт тактов в условиях ограничений. ИИ может считать такты для отдельных инструкций. Но когда тебе нужно знать точную стоимость подпрограммы, которая охватывает спорную и неспорную память, включает условные переходы с разной стоимостью при выполнении/невыполнении условия и должна уложиться в бюджет из 2 340 T-state (одна строка развёртки минус несколько инструкций) — оценки ИИ ненадёжны. Он скажет «примерно 2 200 T-state», когда реальная стоимость зависит от вероятности ветвлений и выравнивания памяти. Вот где DeZog становится незаменимым.

Творческий дизайн эффектов. «Придумай визуальный эффект, который хорошо выглядит и укладывается в 8 000 T-state» — на этот вопрос ИИ ответить не может. Он может реализовать эффект, который ты описал. Он не может его изобрести. Творческое ядро демосценовой работы — поиск вычислительной схемы, производящей захватывающую визуализацию в жёстком бюджете — остаётся полностью человеческим.

23.3 Интеграция с DeZog: вторая половина цикла

Если ИИ генерирует код, DeZog говорит тебе, работает ли он.

DeZog — расширение VS Code, предоставляющее интерфейс отладчика Z80. Оно подключается к эмуляторам (ZEsarUX, CSpect, MAME) или к собственному внутреннему симулятору Z80 и даёт тебе точки останова, инспекцию памяти, наблюдение за регистрами, стеки вызовов и представления дизассемблированного кода — стандартный опыт отладки, который ожидают современные разработчики, применённый к Z80-коду.

Рабочий процесс ИИ + DeZog

Наиболее продуктивный рабочий процесс для Z80-разработки с помощью ИИ объединяет Claude Code с DeZog в плотном цикле:

1. **Claude Code генерирует подпрограмму** — скажем, умножение 8x8.
2. **Ты собираешь её** с помощью mza и загружаешь в эмулятор, подключённый к DeZog.
3. **Ставишь точку останова** на точке входа и пошагово проходишь.
4. **Наблюдаешь за регистрами** на каждом шаге. Содержит ли A правильное промежуточное значение после первого ADD A,B? Устанавливается ли флаг переноса, когда должен?
5. **Замечаешь расхождение** — старший байт результата неверен. Делаешь скриншот состояния регистров или копируешь значения.
6. **Вставляешь расхождение обратно в Claude Code** — «После 6 итераций цикла сдвига A = \$3C, а должен быть \$78. Вот значения регистров в точке останова.»
7. **Claude Code определяет ошибку** — обычно это пропущенный сдвиг, неверный выбор регистра или ошибка на единицу в счётчике цикла.
8. **Исправляешь, пересобираешь, перетестируешь.**

Этот процесс эффективен, потому что даёт ИИ то, чего ему не хватает: наземную истину. ИИ хорош в рассуждениях о структуре кода, но плох в мысленной симуляции выполнения Z80 на протяжении множества итераций. DeZog представляет фактическое состояние выполнения. ИИ рассуждает о разрыве между ожидаемым и фактическим состоянием. Вместе они сходятся к корректному коду быстрее, чем каждый по отдельности.

Инспекция памяти для кода, работающего с данными

Для подпрограмм, которые манипулируют памятью — заполнение экрана, генерация таблиц, операции с буферами — представление памяти в DeZog незаменимо. Ты можешь поставить точку останова после подпрограммы генерации таблицы синусов и проинспектировать 256 байт по адресу таблицы. Симметричны ли они? Достигают ли пика на правильном значении? Пересекают ли ноль в правильной позиции?

Это особенно ценно для сгенерированных ИИ таблиц подстановки. Claude Code может сгенерировать подпрограмму, которая вычисляет 256-байтную таблицу синусов, используя параболическую аппроксимацию из Главы 4. Подпрограмма обычно *почти* работает — форма правильная, диапазон правильный, но может быть ошибка на единицу в индексе, сдвигающая всю таблицу на одну

позицию, или ошибка знака, инвертирующая один квадрант. DeZog позволяет увидеть таблицу напрямую и сравнить с заведомо корректными значениями.

Что DeZog не может делать (пока)

DeZog пока не интегрируется с ИИ-агентами программно. Ты, человек, — мост: читаешь значения регистров, вставляешь их в запрос, применяешь исправления. ИИ-агент, который мог бы ставить точки останова и итерировать автономно, замкнул бы цикл для хорошо определённых задач. Для творческой и архитектурной работы человек остаётся в цикле.

23.4 Когда ИИ помогает, когда нет

Давай будем конкретны. Не «ИИ хорош в некоторых вещах» — конкретные категории с конкретными оценками.

ИИ помогает: высокая уверенность

Кодирование инструкций и подсчёт тактов. ИИ выучил набор инструкций Z80 наизусть: опкоды, размеры в байтах, стоимость в T-state. DJNZ при переходе = 13T, без перехода = 8T. LDIR за байт = 21T, кроме последнего = 16T. Он получает это правильно стабильно, с оговоркой, что иногда путает тайминги Pentagon и 48K со спорной памятью.

Шаблонный код и каркас. Директивы ORG, циклы HALT, записи в регистры AY, подпрограммы очистки экрана, настройка прерываний. Паттерны, виденные тысячи раз. Генерируются корректно, экономят набор текста.

Перевод между диалектами и объяснение кода. Конвертация между синтаксисами sjasmplus, mza и z80asm. Объяснение, что делает блок Z80-ассемблера — отслеживание логики, распознавание паттернов. Чтение Z80 легче, чем написание, и ИИ читает хорошо.

ИИ помогает: средняя уверенность

Стандартные алгоритмы. Умножение сдвигом и сложением, восстановливающее деление, рисование линии Брезенхэма, скроллинг на основе LDIR. ИИ генерирует рабочие реализации этого, но обычно это учебные версии — корректные, но не оптимизированные. Человек выжмет ещё 5–15% скорости через трюки с распределением регистров, использование флагов и разворотку цикла, о которых ИИ не додумается применить.

Раскладка памяти и адресация. «Расположи таблицу размером 256 байт, выровненную по странице, по адресу \$xx00» или «вычисли адрес атрибута для экранной позиции (строка, столбец)». ИИ понимает раскладку экрана Spectrum и генерирует корректные вычисления адресов, хотя иногда ошибается на границе третей в чересстрочной раскладке пиксельной памяти.

Простой самомодифицирующийся код (SMC). Подмена непосредственно операнда, изменение адреса перехода, замена инструкции. ИИ понимает

концепцию и генерирует корректные примеры для простых случаев. Сложная самомодификация — где поведение модифицированного кода зависит от взаимодействия нескольких патчей — ненадёжна.

ИИ не помогает: низкая уверенность

Новаторская оптимизация внутренних циклов. Это главное. Когда тебе нужно сбросить 3 T-state с внутреннего цикла, который выполняется 6 144 раза за кадр — когда 3 T-state — это разница между 50 fps и 48 fps — ИИ не может надёжно найти оптимизацию. Он предложит стандартные подходы (развёртка цикла, таблица подстановки, замена регистров), но не обнаружит конкретный трюк, который позволяет *данная конкретная* раскладка данных и распределение регистров.

Внутренний цикл ротозумера Introspec — `ld a,(hl) : inc l : dec h : add a : add a : add (hl)` — из его анализа Illusion (Глава 7) занимает 95 T-state на 4 пары «толстых» пикселей. Гениальность — в выборе `inc l` вместо `inc hl` (экономия 2 T-state, 6 на пару) и использовании того факта, что `add a` (удвоение) стоит 4T, тогда как `sra a` (сдвиг, делающий то же самое) стоит 8T. Именно такие микрорешения, накапливаясь, создают разницу между демо, которое работает, и демо, которое не работает. ИИ принимает такие решения плохо, потому что они требуют одновременного понимания *глобального* контекста давления на регистры, выравнивания памяти и бюджета кадра.

Тайминги спорной памяти. Паттерн задержки на оригинальных Спектрумах (6, 5, 4, 3, 2, 1, 0, 0 дополнительных T-state за 8-T-state период) взаимодействует с таймингами инструкций так, что ИИ не может надёжно предсказать. Introspec задокументировал это в «GO WEST» (Hype, 2015). ИИ знает факты, но не может применить их для расчёта фактического времени выполнения подпрограмм со смешанным доступом к спорной/неспорной памяти.

Трюки с флагами и эстетические суждения. ИИ знает, что ADD A,A устанавливает перенос из бита 7 — пригодный и как условие перехода, и как умножение — но не комбинирует спонтанно такие факты в новые оптимизации. И он не может принимать творческие решения: какие цвета подойдут для плазмы, каким должно быть ощущение от туннеля, должен ли скроллер подпрыгивать или двигаться синусоидой.

23.5 Кейс-стади: создание MinZ

MinZ — язык программирования для систем Z80 и eZ80, созданный Alice при значительной помощи ИИ в течение 2024–2026 годов. Он компилирует современный, читаемый код в эффективный Z80-ассемблер. Проект реальный, с открытым исходным кодом, на момент написания имеет версию 0.18.0.

MinZ относится к этой главе по двум причинам. Во-первых, это кейс-стади по разработке инструментов для Z80 с помощью ИИ. Во-вторых, это сам по себе пример паттерна HiSoft C — язык более высокого уровня на ограниченном железе, со знакомыми сильными сторонами и пределами.

Что такое MinZ

MinZ предоставляет типизированные переменные (`u8`, `u16`, `i8`, `i16`, `bool`), функции с множественными возвращаемыми значениями, управление потоком (`if/else`, `while`, `for i in 0..n`), структуры, массивы и стандартную библиотеку, охватывающую математику, графику, ввод, звук и операции с памятью. Он компилируется в Z80-ассемблер через собственный ассемблер (`mza`), работает на собственном эмуляторе (`mzx`) и нацелен на ZX Spectrum, CP/M, MSX и Agon Light 2.

Инструментарий включает четыре отдельных инструмента:

- **`mza`** — Z80-ассемблер с макросами, несколькими форматами вывода (`.sna`, `.tap`, `.com`, `.rom`, `.bin`) и поддержкой нескольких платформ
- **`mzx`** — эмулятор ZX Spectrum с безголовым CLI-режимом, автоматическими скриншотами, инъекцией нажатий клавиш и покадровым захватом
- **`mzd`** — Z80-дизассемблер с рекурсивным анализом в стиле IDA, перекрёстными ссылками, подсчётом T-state и реассемблируемым выводом
- **Компилятор MinZ** — компилирует исходный код MinZ в Z80-ассемблер через `mza`

Программа на MinZ выглядит так:

```
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...
    }
}
```

Это компилируется в Z80-ассемблер, собирается в бинарник и работает на реальном или эмулируемом железе. Самодостаточный инструментарий — компилятор, ассемблер, эмулятор, дизассемблер — означает отсутствие внешних зависимостей.

Где ИИ помог при создании MinZ

Сам компилятор. Компилятор MinZ написан на Go (~90 000 строк). Основная часть генерации кода — трансляция промежуточного представления MinZ в Z80-ассемблер — была написана в цикле с помощью ИИ. Паттерн: описать семантику языковой конструкции, сгенерировать генератор кода, протестировать на эмуляторе, исправить расхождения. Для стандартных конструкций вроде арифметических выражений, вызовов функций и управления потоком этот цикл сходился быстро. Claude Code генерировал корректные генераторы кода для `if/else` и циклов `while` с первой или второй попытки.

Ассемблер. `mza`, Z80-ассемблер MinZ, был создан с помощью ИИ. Он поддерживает полный набор инструкций Z80, макросы, множественные форматы

вывода и двухпроходную сборку. Таблица кодирования инструкций — которая отображает мнемоники в опкоды, обрабатывая все нерегулярные префиксные байты Z80 (CB, DD, ED, FD) — была сгенерирована ИИ и верифицирована по даташиту Z80. Это именно тот вид систематического, табличного кода, с которым ИИ справляется хорошо.

Эмулятор. `mzx` достигает 100% покрытия инструкций Z80, включая все недокументированные опкоды (NOP с префиксом ED, DDCB/FDCB индексированные битовые операции). ИИ сгенерировал начальную реализацию каждой инструкции по руководству Z80; тестовый набор (тоже сгенерированный ИИ) поймал крайние случаи — поведение флагов при переполнении, флаг полупереноса в DAA, тайминги прерываний. Но самая полезная возможность `mzx` — целиком построенная через цикл обратной связи с ИИ — это безголовый CLI-режим:

```
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
```

Флаг `--run` загружает бинарник по указанному адресу и запускает выполнение — без ROM, без BASIC, без экрана загрузки. Триггер `--frames DI:HALT` делает скриншот в точный момент, когда код сигнализирует «кадр готов», отключая прерывания перед HALT. Флаг `--dump-keyframes` сохраняет только кадры, в которых экран изменился — автоматический визуальный регрессионный тест. Флаги `--exec` и `--type` инъектируют команды BASIC и нажатия клавиш, позволяя полностью автоматизировать тестирование программ, ожидающих пользовательского ввода.

Конвейер скриншотов этой книги использует `mzx` напрямую. Каждый скриншот примера кода на этих страницах был сгенерирован так:

```
mzx --run build/example.bin@8000 --frames 50 --screenshot build/ch09_plasma.png
```

Двадцать один пример, ноль ручного вмешательства, воспроизводимо через `make screenshots`.

Дизассемблер. `mzd` выполняет рекурсивный анализ — ту же технику, что использует IDA Pro. Получив бинарник, он отслеживает все пути выполнения от точек входа, отделяет код от данных, обнаруживает строки, генерирует перекрёстные ссылки и автоматически расставляет метки на целях переходов:

Флаг `--cycles` добавляет подсчёт T-state к каждой инструкции — автоматизируя ту самую работу, которую Introspec делал вручную в своём разборе Illusion 2017 года. Флаг `--target spectrum` аннотирует системные вызовы (RST \$10 для вывода символа, порт \$FE для бордюра/клавиатуры). Флаг `-R` производит реассемблируемый вывод, замыкая цикл дизассемблирование-модификация-реассемблирование.

ИИ построил и декодер инструкций `mzd` (систематическая табличная работа), и его движок анализа (рекурсивный спуск, построение графа потока управления). Платформо-специфичные знания ABI (что делают какие вызовы ROM ZX Spectrum) были частично сгенерированы ИИ, частично взяты из существующей документации.

Стандартная библиотека и reerhole-оптимизатор. Десять модулей стандартной библиотеки (математика, графика, ввод, звук и т.д.) и 35+ reerhole-паттернов («заменить LD A, 0 на XOR A»). Оба были сгенерированы ИИ и отшлифованы человеком. ИИ знает набор инструкций достаточно хорошо, чтобы предлагать корректные упрощения; человек проверяет семантическую корректность.

Где ИИ не помог при создании MinZ

True Self-Modifying Code (TSMC). Самая отличительная особенность MinZ — TSMC: компилятор может генерировать код, который переписывает собственные инструкции во время выполнения для повышения производительности. Однобайтовая замена опкода (7-20 T-state) вместо последовательности условного перехода (44+ T-state). Концепция TSMC была изобретением Alice, а не ИИ. ИИ не мог бы предложить: «а что если скомпилированный код патчил бы собственные опкоды для изменения поведения во время выполнения?» — потому что эта идея требует одновременного понимания модели компиляции и кодирования инструкций Z80 на уровне, которого ИИ самостоятельно не достигает.

Парсер. MinZ изначально использовал tree-sitter для парсинга, но столкнулся с проблемами нехватки памяти на больших файлах. Замена — рукописный рекурсивно-нисходящий парсер на Go — была спроектирована Alice при консультации с ИИ (GPT-4, o4-mini и Claude были опрошены для архитектурных советов). ИИ-коллеги согласились, что рукописный парсер — правильный подход, и предложили сохранить тестовый корпус tree-sitter. Но фактический дизайн грамматики парсера — как синтаксис MinZ отображается на узлы AST — был человеческой работой. ИИ мог генерировать код парсера для отдельных правил грамматики, но не мог спроектировать саму грамматику.

Распределение регистров для генератора кода. Решать, какие переменные живут в каких регистрах Z80, когда выгружать в память и как обращаться с нерегулярным регистровым файлом Z80 (только определённые регистры могут использоваться с определёнными инструкциями) — это задача удовлетворения ограничений, с которой ИИ справляется плохо. Он генерирует код, который работает, но расточительно использует регистры, делает ненужные сохранения в память и упускает возможности держать горячие значения в регистрах через границы базовых блоков.

Вердикт по MinZ

MinZ не мог бы существовать без помощи ИИ. Огромный объём систематического кода — кодировщик инструкций, эмулятор, движок анализа дизассемблера, стандартная библиотека, reerhole-паттерны — занял бы у одного разработчика годы ручной работы. С помощью ИИ MinZ прошёл путь от концепции до экосистемы из четырёх инструментов примерно за 18 месяцев.

Но интересные особенности MinZ — TSMC, трансформация лямбд в функции с нулевыми накладными расходами, UFCS-диспетчеризация методов, триггер DI:HALT в mzх, платформо-зависимые ABI-аннотации mzd — это человеческие изобретения. ИИ их реализовал, но не придумал.

Это точно соответствует паттерну HiSoft C. Инструмент колossalльно ускоряет

рутинную работу. Творческая работа остаётся человеческой. Компромисс реален и стоит того.

23.5b Врезка: Другой ИИ — супероптимизация полным перебором

Peephole-оптимизатор MinZ знает 35+ паттернов вроде «заменить LD A, 0 на XOR A». Но как находить такие паттерны? И откуда знать, какие из них действительно безопасны?

Возьмём LD A, 0 → XOR A. Обе инструкции обнуляют А. Обе занимают меньше байтов в форме XOR (1 байт против 2). Но XOR A сбрасывает флаг переноса и устанавливает флаг нуля; LD A, 0 сохраняет все флаги. Если код после этой инструкции проверяет перенос, «оптимизация» — это баг. Человек-эксперт это знает. ИИ на нейросетях *обычно* это знает, но иногда забывает. Супероптимизатор полным перебором *доказывает* это, тестируя каждое возможное входное состояние.

z80-optimizer (от oisee, 2025) доводит подход полного перебора до логического завершения. Он перебирает каждую пару инструкций Z80 — все 406 опкодов × 406 опкодов = 164 836 пар — и для каждой пары проверяет, производит ли более короткая замена идентичный результат для всех возможных состояний регистров и флагов. Никаких эвристик. Никаких обучающих данных. Никаких нейросетей. Только исчерпывающий перебор с полной верификацией эквивалентности состояний.

Результаты: **602 008 доказуемо корректных правил оптимизации** за один запуск на Apple M2 (34,7 миллиарда сравнений за 3 часа 16 минут). Несколько примеров:

Исходная последовательность	Замена	Экономия
SLA A : RR A	OR A	3 байта, 12T
LD A, 0 : NEG	SUB A	2 байта
LD A, B : ADD A, 0	LD A, B : OR A	0 байтов, 4T
SCF : RR A	SCF : RRA	1 байт, 4T

Правила группируются в **83 уникальных паттерна трансформации** — семейства замен, разделяющих одну и ту же структурную логику. Например, семейство «загрузить-и-проверить»: LD A, r : ADD A, 0 → LD A, r : OR A применяется ко всем регистрам-источникам, потому что оптимизация эксплуатирует поведение флагов, а не конкретный регистр.

Что делает z80-optimizer интересным для этой главы — не конкретные правила: любой опытный Z80-кодер знает большинство распространённых. Интересна *методология*. Это ИИ в исходном смысле: машина, которая находит знания через поиск, а не через заученные паттерны. Среди 602 008 правил — тысячи, которые ни один человек не каталогизировал, потому что они связаны с малоизвестными парами опкодов, которые никто не пишет намеренно, но которые компиляторы и генераторы кода производят.

Очевидный следующий шаг — последовательности длины 3 — требует GPU-перебора ($406^3 = 67$ миллионов троек \times все входные состояния). За этим пределом стохастический поиск (в стиле STOKE) позволяет исследовать пространство более длинных замен без исчерпывающего перебора.

Для практической Z80-разработки z80-optimizer дополняет цикл обратной связи с ИИ из этой главы: Claude Code генерирует корректный, но неоптимизированный код, а затем z80-optimizer может механически проверить, есть ли у каких-либо пар инструкций более короткие эквиваленты. Один ИИ пишет код; другой ИИ доказывает, как его ужать.

Исходный код: github.com/oisee/z80-optimizer (лицензия MIT)

23.6 Честный взгляд: «Z80 они по-прежнему не знают»

Скептицизм Introspec в отношении Z80-возможностей ИИ — это не общая технофобия. Он исходит из десятилетий опыта выжимания Z80 до абсолютного предела. Когда он говорит «Z80 они по-прежнему не знают», он имеет в виду кое-что конкретное.

Рассмотрим внутренний цикл ротозумера из его анализа Illusion. Эффект проходит через текстуру под углом, создавая повёрнутые и масштабированные «толстые» пиксели 2x2. Внутренний цикл:

```
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
```

Ключевое наблюдение — inc l вместо inc hl. Это экономит 2 T-state, но работает только потому, что текстура выровнена по 256-байтной границе, так что инкремент одного L никогда не требует переноса в H. ИИ использовал бы inc hl — безопасный, универсальный выбор — и потерял бы 2 T-state на итерацию. При 3 072 парах пикселей за кадр это 6 144 T-state — почти 10% бюджета кадра на 48K Spectrum.

Вот что означает «Z80 они по-прежнему не знают». Не то, что ИИ не может писать Z80-ассемблер — может. Не то, что он путает инструкции — обычно нет. А то, что он не понимает *контекст*, в котором каждый выбор инструкции имеет значение. Он не мыслит бюджетами. Он не видит, что inc l безопасен здесь из-за ограничения выравнивания тремя уровнями абстракции выше. Он не знает, что 2 T-state на итерацию, умноженные на тысячи итераций — это разница между попаданием в один кадр и необходимостью двух.

Что именно ИИ делает неправильно

Мы протестировали Claude Code на нескольких Z80-задачах из области демосцены. Вот показательные результаты.

Задача: написать подпрограмму DOWN_HL. DOWN_HL перемещает указатель экрана в HL на одну строку пикселей вниз в чересстрочной раскладке видеопамяти Spectrum. Это самый часто используемый графический примитив на платформе. Introspec написал об этом фундаментальную статью (Нуре, 2020).

Первая попытка ИИ:

```
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
```

Это действительно корректный код — он правильно обрабатывает все три трети экрана, включая переходы между ними. ИИ может правильно реализовать стандартные подпрограммы вроде DOWN_HL, потому что паттерн хорошо задокументирован и встречается во многих Z80-справочниках. Подпрограмма работает: `inc h` переходит к следующей строке пикселей, `add a, 32` переходит к следующему знакоряду при необходимости, перенос от сложения L корректно определяет границы третей, а `sub 8` отменяет лишний инкремент ТТ для обычного случая.

Но «корректно» — не то же самое, что «хорошо». В статье Introspec представлена версия RST7, использующая подход с двойным счётчиком, которая обрабатывает все границы за 2 343 T-state при полном проходе по экрану. Наивный подход выше — стандартная учебниковая версия — стоит 5 922 T-state. Разрыв между «работает» и «работает хорошо» — в 2,5 раза, и ИИ этот разрыв не преодолевает. Он выдаёт первую версию, которую написал бы любой грамотный программист, а не ту, которую эксперт оптимизировал бы до совершенства.

Задача: сгенерировать развёрнутое заполнение экрана. На просьбу сгенерировать развёрнутое заполнение экрана на основе PUSH (техника из Главы 3) ИИ произвёл корректный код — пары PUSH, записывающие по два байта, DI/EI для защиты манипуляции с указателем стека. Но он не подумал расположить данные в обратном порядке (PUSH записывает старший байт первым, по более низким адресам), из-за чего паттерн заполнения оказался задом наперёд. Человек, который писал PUSH-заполнения раньше, учитывает это автоматически.

Задача: оптимизировать заданный внутренний цикл. Получив работающий внутренний цикл с просьбой ускорить его, ИИ предложил стандартные оптимизации: развёртку, таблицу подстановки, замену регистров. Это корректные предложения. Но он не нашёл неочевидную оптимизацию — ту, где ты перестраиваешь раскладку памяти, чтобы использовать `inc l` вместо `inc hl`,

или используешь флаг переноса от сложения как условие перехода вместо отдельного сравнения. Неочевидная оптимизация требует понимания полного контекста подпрограммы, а контекстное окно ИИ, хоть и большое, не схватывает *пространственную и временную* структуру Z80-программы так, как это делает ментальная модель человека-эксперта.

В чём Introspec прав

Глубочайшие оптимизации Z80 — это знание инструкций. Это понимание взаимодействия между раскладкой памяти, распределением регистров, кодированием инструкций, ограничениями по таймингам и визуальным выводом — одновременно. Это взаимодействие и есть то, что Introspec называет «эволюцией вычислительной схемы» (Глава 1). Вычислительная схема — это целостный дизайн, где каждое решение влияет на все остальные. ИИ оперирует кодом локально. Эксперт оперирует схемой глобально.

ИИ не знает Z80 в том смысле, в каком Introspec знает Z80. Он выучил набор инструкций наизусть, но не проникся машиной.

В чём Introspec не совсем прав

Но «Z80 они по-прежнему не знают» подразумевает, что ИИ бесполезен для Z80-работы, а это не так.

ИИ не пытается заменить Introspec. Он пытается помочь Alice — программисту, который понимает Z80 достаточно хорошо, чтобы оценивать вывод ИИ, но не имеет десятилетий опыта оптимизации внутренних циклов. Для Alice вывод ИИ — это стартовая точка, которая лучше, чем пустой экран. Ей не нужно, чтобы ИИ нашёл трюк с `inc l`. Ей нужно, чтобы он сгенерировал первые 80% подпрограммы, и тогда она может потратить время на оставшиеся 20%.

Демосцена всегда была про последние 20%. ИИ это не меняет. Он меняет скорость, с которой ты проходишь первые 80%.

23.7 Демо «Antique Toy»: ИИ на практике

Демо-компаньон этой книги — «Antique Toy» — это сознательный эксперимент: построить демо для ZX Spectrum с помощью ИИ и задокументировать, что произойдёт.

Название — отсылка к *Eager Introspec* (2015, 1-е место на ЗВМ openair). Мы реализуем эффекты, вдохновлённые Eager — атрибутный туннель с 4-кратной симметрией, хаотический зумер, анимация 4-фазным цветом — плюс 3D-движок на методе средней точки от Dark из *Spectrum Expert #02*.

Что сработало: Прототипирование эффектов — Claude Code генерирует рабочие первые черновики достаточно быстро, чтобы пробовать идеи, которые иначе не стоили бы времени на набор. «А что если туннель использовал 8-кратную симметрию вместо 4-кратной?» занимает 15 минут со сгенерированным ИИ кодом вместо 2 часов вручную. Инструменты — система сборки, пайплайн ресурсов, правила Makefile и тестовые обвязки были полностью сгенерированы ИИ и работают надёжно. Ревью кода — подача ИИ подпрограммы с вопросом

«что тут не так?» ловит очевидные ошибки (ошибки на единицу, забытые DI/EI, неправильные номера портов) до того, как они обойдутся в часы отладки.

Что не сработало: 3D-движок по методу средней точки Dark. Виртуальный процессор с упакованными 2-битными опкодами и 6-битными номерами точек был декодирован неправильно. Инструкция усреднения вычисляла $(A+B)/2$ через ADD A,B : SRA A, что переполняется для знаковых координат. Три сеанса отладки, дольше, чем написать с нуля. Интеграция музыки провалилась аналогично — ИИ сгенерировал проигрыватель, который конфликтовал с использованием теневых регистров (EXX, EX AF,AF') кодом эффекта. И проигрыватель, и эффект использовали теневой BC для разных целей, и EXX в обработчике прерываний подставлял устаревшие значения. Этот класс ошибок — системные конфликты регистров через границы прерываний — требует понимания полной архитектуры системы, а не отдельных подпрограмм.

Честная оценка: «Antique Toy» не закончен. Эффекты работают по отдельности. Интеграция продолжается. Но помошь ИИ сделала проект *осуществимым* для сольного разработчика, работающего по вечерам и выходным. Правильный вопрос не «может ли ИИ сравняться с выделенной человеческой командой?», а «позволяет ли помошь ИИ большему количеству людей делать демо?» Ответ, предварительно, да.

23.8 Цикл обратной связи на практике

Конкретный пример из проекта «Antique Toy»: реализация 4-кратной симметрии для эффекта туннеля путём копирования верхнего левого квадранта атрибутов 16x12 в три остальных квадранта с зеркалированием.

Запрос был конкретным: «Напиши Z80-подпрограмму, которая копирует верхний левый квадрант 16x12 области атрибутов ZX Spectrum (\$5800) в три остальных квадранта с соответствующим зеркалированием.» Claude Code сгенерировал 47 строк, которые собрались с первой попытки.

Тестирование показало, что верхний правый квадрант смешён на один столбец. DeZog показал проблему: после того как зеркальный цикл декрементировал DE 16 раз, вычисление перехода к следующей строке забыло, что DE уже был сдвинут назад. Код продвигал DE на 32 (ширина одной строки) вместо необходимых 48 (32 для строки + 16 для компенсации зеркального прохода). Вставка значений регистров в Claude Code — «После строки 1 DE = \$581F (должно быть \$582F)» — дала немедленное исправление. Правый нижний квадрант имел ту же ошибку в усиленном виде. Ещё одна итерация исправила его.

Итого: три итерации, примерно 25 минут. Ручная оценка для опытного Z80-программиста: 40–60 минут. Для новичка: 2–3 часа. ИИ сэкономил время на начальной генерации. Отладка заняла одинаковое время независимо от того, кто написал код.

23.9 Построение собственного рабочего процесса с ИИ

Практическая настройка: VS Code с расширением Z80 Macro Assembler и Z80 Assembly Meter. Claude Code (или любая LLM, способная работать с кодом). Ассемблер (`mza` или `sjasmlplus`). DeZog, подключённый к эмулятору. Makefile.

Рабочий процесс: **Начни с ИИ** — опиши, что хочешь, с конкретикой (целевая машина, адреса памяти, синтаксис ассемблера). **Собери немедленно** — не читай внимательно код ИИ; собери его, вставь ошибки обратно. **Тестируй цветом бордюра** — оберни сгенерированные ИИ подпрограммы в тестовую обвязку из Главы 1. **Отлаживай с DeZog** — ставь точки останова, найди первое расхождение регистров, сообщи его ИИ. **Итерируй** — обычно 2-5 раундов для средней сложности; больше 5 означает, что ИИ не справляется, и тебе стоит написать самостоятельно. **Оптимизируй сам** — после достижения корректности профилируй и применяй техники из Глав 1-14.

Промпт-инжиниринг для Z80

Хороший запрос: «Напиши Z80-подпрограмму для ZX Spectrum 128K (тайминги Pentagon), которая копирует 16 байт из адреса в HL в экранную память по адресу (DE), следуя чересстрочной раскладке Spectrum. После каждого байта продвигай DE на следующую строку пикселей стандартным методом `down_hl`. Используй синтаксис `mza`. Добавь подсчёт тактов.»

Плохой запрос: «Напиши подпрограмму для спрайтов на Spectrum.»

Хороший запрос указывает машину, ассемблер, адреса, поведение и формат вывода. Плохой запрос оставляет всё неоднозначным, и ИИ заполнит пробелы неверными предположениями.

Для запросов на оптимизацию давай конкретную цель: «Эта подпрограмма занимает ~3 200 T-state. Мне нужно меньше 2 400. Не меняй интерфейс (HL = источник, DE = назначение, B = высота). Тайминги Pentagon.» Целевая производительность и ограничение интерфейса заставляют ИИ искать реальные оптимизации вместо реструктуризации соглашения о вызовах.

23.10 Общая картина

Помощь ИИ не меняет уровень абстракции результата — Z80 по-прежнему выполняет те же инструкции на тех же скоростях. Что она меняет — скорость ввода: как быстро ты переходишь от идеи к работающему (пусть и неоптимизированному) коду. Эксперты демосцены по-прежнему будут писать внутренние циклы лучше любого ИИ, но инструменты с ИИ-помощью снижают порог входа настолько, что больше людей смогут начать делать демо и самостоятельно освоить глубинные трюки.

Итого

- **Z80-разработка с помощью ИИ** следует циклу обратной связи: запрос, генерация, сборка, тест, отладка, итерация. ИИ генерирует первый черновик быстро; человек оценивает и доводит. Цикл обычно занимает 2–5 итераций для подпрограммы средней сложности.
- **ИИ надёжен для:** кодирования инструкций, подсчёта тактов, шаблонного кода, перевода между диалектами и объяснения кода. Умеренно надёжен для стандартных алгоритмов и простого самомодифицирующегося кода (SMC). Ненадёжен для новаторской оптимизации, таймингов спорной памяти, творческого дизайна эффектов и глубоких трюков с флагами.
- **Интеграция с DeZog** ликвидирует разрыв между выводом ИИ и корректным кодом. Человек считывает состояния регистров из отладчика и передаёт расхождения ИИ, который рассуждает о несоответствии. Программной интеграции ИИ с отладчиком пока не существует, но это очевидный следующий шаг.
- **Кейс-стади MinZ** ясно демонстрирует паттерн: помощь ИИ позволила одному разработчику построить полный инструментарий языка (компилятор, ассемблер, эмулятор, стандартная библиотека) за 18 месяцев. Рутинная работа — кодирование инструкций, генерация тестов, функции стандартной библиотеки — была сгенерирована ИИ. Творческая работа — TSMC, абстракции с нулевыми накладными расходами, дизайн грамматики — была человеческой.
- **Скептицизм Introspec обоснован:** ИИ не понимает Z80 так, как понимает эксперт. Он не мыслит бюджетами, не видит сквозных ограничений, не находит неочевидных оптимизаций. Глубочайшая демосценовая работа остаётся за пределами досягаемости ИИ.
- **Историческая параллель верна:** HiSoft C был «в 10–15 раз быстрее BASIC», но не имел чисел с плавающей точкой. Z80-разработка с помощью ИИ драматически быстрее для шаблонного кода и итераций, но не может сравниться с человеком-экспертом в оптимизации внутренних циклов. Инструменты более высокого уровня на ограниченном железе всегда были компромиссом. Вопрос не «хорошо или плохо?», а «хорошо для чего?»
- **Практический рабочий процесс** сочетает Claude Code для генерации кода, DeZog для отладки, mza или sjasmplus для сборки и Makefile для автоматизации сборки. Начни с ИИ, собери немедленно, протестируй цветом бордюра, отлаживай с DeZog, оптимизируй сам.
- **Общий эффект** положительный: помощь ИИ снижает порог входа в Z80-разработку, не снижая потолок. Больше людей могут начать; эксперты по-прежнему нужны для глубокой работы. Это хорошо для демосцены.

Попробуй сам

1. **Тест шаблонного кода.** Попроси своего ИИ-ассистента сгенерировать шаблон загрузки ZX Spectrum 128K: ORG по адресу \$8000, отключение

прерываний, настройка IM1, цикл HALT с тестовой обвязкой цвета бордюра. Собери и запусти. Сколько итераций понадобилось?

2. **Тест оптимизации.** Напиши (или сгенерируй ИИ) работающий цикл заполнения атрибутов. Измерь его стоимость тестовой обвязкой цвета бордюра. Затем попроси ИИ ускорить его. Измерь снова. Теперь оптимизируй его сам, используя техники из Глав 1-3. Сравни все три версии: оригинальную, оптимизированную ИИ, оптимизированную человеком.
3. **Испытание DOWN_HL.** Попроси ИИ написать подпрограмму DOWN_HL. Протестируй её на всех 192 строках пикселей. Правильно ли она обрабатывает переходы между третями? Сравни с анализом Introspec (Hype, 2020). Это лакмусовый тест компетенции ИИ в Z80.
4. **Эксперимент с MinZ.** Установи инструментарий MinZ (`mza`, `mzx`, `mzd`). Ассемблируй заливку экрана с помощью `mza`, запусти её в безголовом режиме через `mzx --run fill.bin@8000 --frames 5 --screenshot fill.png`, затем дезассемблируй бинарник демо с помощью `mzd demo.bin --analyze --cycles --target spectrum`. Сравни подсчёт T-state, сделанный ИИ-дезассемблером, со своими собственными ручными подсчётом из Главы 1.
5. **Автоматизированный конвейер.** Напиши эффект, ассемблируй его и добавь в `Makefile`, который запускает `mzx --screenshot` для каждого бинарника. Запусти `mzx --dump-keyframes`, чтобы увидеть, какие именно кадры дают видимые изменения. Это тот самый конвейер, который сгенерировал каждый скриншот в этой книге.
6. **Построй что-нибудь.** Выбери эффект из одной из предыдущих глав. Используй помощь ИИ для написания первого черновика. Итерируй до работающего результата. Профилируй его. Оптимизируй внутренний цикл вручную. Задокументируй каждый шаг. Ты только что прошёл через тот самый рабочий процесс, который описывает вся эта глава.

Это последняя техническая глава. Далее следуют приложения — справочные таблицы, руководства по настройке и справочник инструкций, к которому ты будешь обращаться каждый раз, когда пишешь Z80-ассемблер.

Источники: Обзор HiSoft C (Spectrum Expert #02, 1998); Introspec «Technical Analysis of Illusion» (Hype, 2017); Introspec «DOWN_HL» (Hype, 2020); Introspec «GO WEST Parts 1-2» (Hype, 2015); z80-optimizer (oisse, 2025, github.com/oisse/z80-optimizer)

Глоссарий

Техническая лексика, используемая в книге «Кодируя невозможное». Термины сгруппированы по категориям; «Впервые» указывает главу, где термин вводится или определяется, «Также» перечисляет главы со значительным использованием.

А. Тайминг и производительность

Термин	Определение	Каноническая форма	Впервые	Также
Такт (T-state)	Один тактовый цикл процессора Z80 на частоте 3,5 МГц (~286 нс). Фундаментальная единица времени выполнения на Spectrum.	«такты (T-state)» в тексте; «T» в комментариях кода (напр., ; 11T)	Гл01	Гл02-Гл23
Кадр	Одно полное обновление экрана на частоте ~50 Гц (PAL). Длительность варьируется по моделям машин.	«кадр»	Гл01	все

Термин	Определение	Каноническая форма	Впервые	Также
Бюджет кадра	<p>Общее число тактов (T-state) между прерываниями:</p> <p>Pentagon 71 680, ZX 128K 70 908, ZX 48K 69 888.</p> <p>Практический бюджет после HALT + ISR + проигрыватель музыки: ~66 000-68 000 (Pentagon), ~55 000-60 000 (128K с записью в экран во время активного отображения).</p> <p>Подробнее в Гл15 – тakt-карты с разбивкой по областям (верхний бордюр, активное отображение, нижний бордюр).</p>	«бюджет кадра»	Гл01	Гл04, Гл05, Гл08, Гл10, Гл12, Гл14, Гл16, Гл17, Гл18, Гл21
Строка развёртки	<p>Одна горизонтальная линия дисплея.</p> <p>Ширина варьируется: 224 такта (T-state) на 48K/Pentagon, 228 тактов на 128K. Кадр состоит из 320 (Pentagon), 311 (128K) или 312 (48K) строк развёртки, включая бордюры.</p>	«строка развёртки» (одно слово)	Гл01	Гл02, Гл08, Гл15, Гл17

Термин	Определение	Каноническая форма	Впервые	Также
Спорная память	Страницы ОЗУ (\$4000–\$7FFF), где ULA и процессор делят шину памяти на Sinclair-оборудовании. Обращения процессора задерживаются на 0–6 дополнительных тактов (T-state) за обращение в повторяющемся 8-T-state паттерне. Клоны Pentagon не имеют конкуренции.	«спорная память»	Гл01	Гл04, Гл05, Гл15, Гл17, Гл18, Гл21, Гл22, Гл23
Машинный цикл (M-цикл)	Группа из 3–6 тактов (T-state) внутри инструкции. Первый M-цикл (M1) – всегда выборка опкода за 4 такта.	«машинный цикл» или «M-цикл»	Гл01	-
Время бордюра	Строки развёртки за пределами 192-строчного активного дисплея (верхний/нижний бордюры). Без конкуренции; ~14 000 тактов (T-state) доступно на 128K.	«время бордюра»	Гл01	Гл08, Гл15, Гл17

Термин	Определение	Каноническая форма	Впервые	Также
Тестовая обвязка	Отладочная техника: установить цвет бортюра в красный перед кодом, в чёрный после; высота полоски на экране показывает стоимость в тактах (T-state).	«тестовая обвязка с цветом бордюра»	Гл01	Гл02, Гл03, Гл08, Гл18

В. Аппаратура — Sinclair и клоны

Термин	Определение	Каноническая форма	Впервые	Также
Z80	Процессор Zilog Z80A на частоте 3,5 МГц. 8-битная шина данных, 16-битная шина адреса. Процессор во всех моделях ZX Spectrum.	«Z80»	Гл01	все
ULA	Uncommitted Logic Array. Заказная микросхема, генерирующая видеосигнал и обрабатываю- щая ввод/вывод (клавиатура, магнитофон, динамик). Делит шину памяти с процессором на Sinclair- оборудовании.	«ULA»	Гл01	Гл02, Гл08, Гл15

Термин	Определение	Каноническая форма	Впервые	Также
ZX Spectrum 48K	Оригинальная модель Sinclair. 69 888 тактов (T-state)/кадр, 312 строк развертки, спорная нижняя 32K.	«48K» или «ZX Spectrum 48K»	Гл01	Гл11, Гл15
ZX Spectrum 128K	Расширенная модель со 128K банковой ОЗУ, звуковым чипом AY, двумя страницами экрана. 70 908 тактов (T-state)/кадр, 311 строк развертки.	«128K» или «ZX Spectrum 128K»	Гл01	Гл11, Гл15, Гл20, Гл21
Экран-ная память	\$4000-\$5AFF. Пиксельные данные (\$4000-\$57FF, 6 144 байта) + область атрибутов (\$5800-\$5AFF, 768 байт). Чресстрочная раскладка строк пикселей.	«экранная память»	Гл01	Гл02, Гл08, Гл16, Гл17
Область атрибу-тов	768 байт по адресу \$5800-\$5AFF. Один байт на 8x8 ячейку атрибутов: FBPPPIII (Flash, Bright, Paper x3, Ink x3). Управляет цветом.	«область атрибутов»	Гл02	Гл08, Гл09, Гл10

Термин	Определение	Каноническая форма	Впервые	Также
Конфликт атрибутов	Аппаратное ограничение: только 2 цвета (чернила + фон) на 8x8 ячейку. Перекрывающиеся спрайты вынуждают идти на цветовые компромиссы.	«конфликт атрибутов»	Гл02	Гл08, Гл16
Чересстрочная раскладка экрана	Строки пикселей в видеопамяти расположены не последовательно. Адрес кодирует Y как 010TTSSS.LLLCCCCC в двух байтах. Три 2 048-байтные трети.	«чересстрочная раскладка экрана»	Гл02	Гл07, Гл16, Гл17, Гл22
Теневой экран	Вторая страница экрана по адресу \$C000 (банк 7) на 128K. Переключается через порт \$7FFD, бит 3.	«теневой экран»	Гл08	Гл10, Гл15, Гл21
AY-3-8910	Программируемый звуковой генератор General Instrument. Три канала прямоугольного тона, один генератор шума, один генератор огибающей. 14 регистров. Стандарт на моделях 128K.	«AY-3-8910» (первое упоминание), затем «AY»	Гл11	Гл12, Гл15, Гл21
Порт \$FFFD / BFFD /AY128K. «FFFD» / «\$BFFD»	Гл11	Гл12		

Термин	Определение	Каноническая форма	Впервые	Также
Порт \$7FFD	Порт переключения банков памяти и выбора экрана на 128K.	«\$7FFD»	Гл08	Гл12, Гл15, Гл21
Порт $FE / :$ $0--2 =$ $,3 =$ $MIC,4 =$ $EAR/.(-$). «FE»	Гл01	Гл02, Гл11, Гл18		
IM1 / IM2	Режимы прерываний. IM1: обработчик по адресу \$0038. IM2: векторный через регистр I + байт шины данных; используется для пользова- тельских обработчиков прерываний (проигрыватели музыки, много- поточность).	«IM1» / «IM2»	Гл01	Гл03, Гл05, Гл11, Гл12
DivMMC	Интерфейс SD-карты для современного использования Spectrum. Поддерживает esxDOS.	«DivMMC»	Гл15	Гл20
ZX Spectrum Next	FPGA- платформа на базе Spectrum. Процессор Z80N (дополнитель- ные инструкции), Triple AY (3xAY), аппаратные спрайты, copper- сопроцессор, тайлмап, турбо 28 МГц.	«ZX Spectrum Next» или «Next»	Гл11	Гл15

С. Аппаратура — советская/постсоветская экосистема

Термин	Определение	Каноническая форма	Впервые	Также
Pentagon 128	Самый популярный советский клон ZX Spectrum. Без спорной памяти, 71 680 тактов (T-state)/кадр, 320 строк развёртки. Эталонная платформа для демосцены ZX Spectrum.	«Pentagon 128» (первое упоминание), затем «Pentagon»	Гл01	Гл04, Гл05, Гл08, Гл11, Гл15, Гл16, Гл17, Гл18, Гл19, Гл20, Гл21, Гл22
Scorpion ZS-256	Советский клон с TurboSound и расширенной памятью. Тайминги, совместимые с Pentagon.	«Scorpion ZS-256» (первое упоминание), затем «Scorpion»	Гл04	Гл11, Гл15, Гл20
TurboSound	Два чипа AY (2xAY) в одной машине, обеспечивая 6 звуковых каналов. Стандарт на Scorpion; доступен как дополнение для Pentagon.	«TurboSound»	Гл11	Гл15, Гл20

Термин	Определение	Каноническая форма	Впервые	Также
TR-DOS	Дисковая операционная система на советских клонах через интерфейс Beta Disk 128. Формат файлов: .trd. Стандартный формат распространения для компо демосцены и дисковых журналов.	«TR-DOS»	Гл15	Гл20
Beta Disk 128	Интерфейс контроллера дискет, стандартный на клонах Pentagon и Scorpion.	«Beta Disk 128»	Гл15	Гл20

D. Аппаратура — Agon Light 2

Термин	Определение	Каноническая форма	Впервые	Также
Agon Light 2	Одноплатный компьютер с процессором Zilog eZ80 на частоте 18,432 МГц и VDP на базе ESP32. 512КБ линейной ОЗУ, без банковости.	«Agon Light 2» или «Agon»	Гл01	Гл11, Гл15, Гл18, Гл22

Термин	Определение	Каноническая форма	Впервые	Также
eZ80	Процессор Zilog eZ80. Совместимый набор инструкций Z80; большинство инструкций выполняется за меньшее число тактов. 24-битная линейная адресация (режим ADL). ~368 640 тактов (T-state)/кадр при 50 Гц.	«eZ80»	Гл01	Гл15, Гл22
VDP	Video Display Processor. Микроконтроллер ESP32 с библиотекой FabGL. Управляет дисплеем, спрайтами, тайлмапами, синтезом звука. Связывается с eZ80 по последовательному интерфейсу на скорости 1 152 000 бод.	«VDP»	Гл02	Гл11, Гл15, Гл18, Гл22
Режим ADL	24-битный режим адресации на eZ80. Линейное 512КБ адресное пространство, без банковости.	«режим ADL»	Гл15	Гл22

Термин	Определение	Каноническая форма	Впервые	Также
MOS	Операционная система на Agon. Представляет API-вызовы для команд VDP, клавиатуры, файловой системы. <code>waitvblank</code> заменяет HALT Spectrum для синхронизации кадров.	«MOS»	Гл18	Гл22

E. Техники

Термин	Определение	Каноническая форма	Впервые	Также
Самомодифицирующийся код (SMC)	Запись в байты инструкций во время выполнения. Безопасно на Z80 (нет кэша инструкций). Типичные паттерны: подмена непосредственных операндов, изменение целей переходов, замена опкодов.	«самомодифицирующийся код (SMC)» при первом упоминании; «SMC» далее	Гл03	Гл06, Гл07, Гл09, Гл10, Гл13, Гл16, Гл17, Гл22, Гл23
Развёрнутый цикл	Обмен размера кода на скорость путём повторения тела цикла N раз, исключая накладные расходы счётчика цикла. Частичная развёртка сохраняет DJNZ для внешнего счётчика.	«развёрнутый цикл»	Гл02	Гл03, Гл07, Гл10, Гл16, Гл17

Термин	Определение	Каноническая форма	Впервые	Также
PUSH-трюк	Перехват SP для использования PUSH как быстрой записи в память (5,5 тактов/байт против 21 для LDIR). Требуется DI для защиты от прерываний, использующих стек.	«PUSH-трюк» или «вывод через стек»	Гл03	Гл07, Гл08, Гл10, Гл16
LDI-цепочка	Последовательность отдельных инструкций LDI; на 24% быстрее LDIR для копий известного размера. Комбинируется с арифметикой точки входа для копий переменной длины.	«LDI-цепочка»	Гл03	Гл07, Гл09
LDPUSH	Внедрение данных экрана в исполняемый код LD DE,nn : PUSH DE (21T на 2 байта). Используется в мультиколор-движках.	«LDPUSH»	Гл08	-
DOWN_HL	Классическая подпрограмма продвижения HL на одну строку пикселей вниз в чересстрочной экранной памяти Spectrum. 20T в типичном случае, 77T в худшем (граница трети).	«DOWN_HL»	Гл02	Гл16, Гл17, Гл23

Термин	Определение	Каноническая форма	Впервые	Также
RET-цепочка	Установить SP на таблицу адресов; каждая подпрограмма заканчивается RET, диспетчеризуя к следующей. 10Т за диспетчериzacию против 17Т для CALL.	«RET-цепочка»	Гл03	-
Генерация кода	Запись машинного кода в буфер во время выполнения с последующим исполнением. Устраняет ветвления и счётчики циклов из внутренних циклов.	«генерация кода»	Гл03	Гл06, Гл07, Гл09
Скомпилированные спрайты	Спрайты, скомпилированные в последовательность инструкций PUSH/LD с предзагруженными регистровыми парами. Фиксированное изображение, максимальная скорость.	«скомпилированные спрайты»	Гл03	Гл16
Двойная буферизация	Поддержание двух страниц экрана для предотвращения разрывов. На 128K: Экран 0 (4000)1(C000), переключаемые через порт \$7FFD.	«двойная буферизация»	Гл05	Гл08, Гл10, Гл12

Термин	Определение	Каноническая форма	Впервые	Также
Грязные прямо- угольни- ки	Сохранение/вос- становление фона под спрайтами до/после рисования. Избегает полной очистки экрана.	«грязные прямоугольники»	Гл08	Гл16, Гл21
Мульти- колор	Изменение значений атрибутов между чтениями строк развёртки ULA для отображения более 2 цветов на 8x8 ячейку. Потребляет 80–90% процессора.	«мультиколор»	Гл08	-
Страниц- ная таблица	256-байтная таблица подстановки, размещённая по адресу \$xx00, чтобы H содержал базу, а L – индекс. Индексация одним регистром с нулевыми накладными расходами.	«страничная таблица»	Гл04	Гл06, Гл07, Гл09, Гл10
Таблица подста- новки	Предвычислен- ная таблица значений для быстрого доступа во время выполнения. Избегает дорогих вычислений во внутренних циклах.	«таблица подстановки»	Гл03	Гл04, Гл07, Гл09, Гл17, Гл19, Гл20, Гл22

Термин	Определение	Каноническая форма	Впервые	Также
Раздельные счётчики	Реструктуризация перебора экрана в соответствии с трёхуровневой иерархией (треть/знакоряд/строка развортки), устраняющая ветвления. На 60% быстрее наивного обхода DOWN_HL.	«раздельные счётчики»	Гл02	-
4-фазный цвет	4-кадровый цикл (2 нормальных + 2 инвертированных атрибута) на 50 Гц. Инерция зрения усредняет цвета, создавая дополнительные воспринимаемые цвета на ячейку.	«4-фазный цвет»	Гл10	-
Цифровые барабаны	Цифровой PCM-сэмпл, воспроизводимый через регистр громкости АУ как 4-битный ЦАП (фаза атаки), с переходом на огибающую АУ (фаза затухания). Потребляет ~2 кадра процессорного времени за удар.	«цифровые барабаны» или «гибридные барабаны»	Гл12	-

Термин	Определение	Каноническая форма	Впервые	Также
Асинхронная генерация визуальных кадров от кадров	Отвязка генерации кадров от отображения через кольцевой буфер. Генератор записывает кадры вперёд; дисплей читает на стабильных 50 Гц. Поглощает процессорные всплески от воспроизведения барабанов.	«асинхронная генерация кадров»	Гл12	-

F. Ассемблерная нотация и директивы

Термин	Определение	Каноническая форма	Впервые	Также
ORG	Директива ассемблера, устанавливающая адрес начала кода. ORG \$8000 для примеров на Spectrum.	ORG	Гл01	все примеры кода
EQU	Определение именованной константы. SCREEN EQU \$4000.	EQU	Гл02	все примеры кода
DB / DW / DS	Define Byte / Define Word / Define Space. DB -3 (отрицательные значения допустимы в sjasmplus).	DB, DW, DS	Гл04	все примеры кода
ALIGN	Выравнивание по границе степени двойки. Директива sjasmplus.	ALIGN	Гл07	-

Термин	Определение	Каноническая форма	Впервые	Также
INCLUDE / INCBIN	Включение файла исходного кода / включение двоичных данных. Директивы sjasmplus.	INCLUDE / INCBIN	Гл14	Гл20, Гл21
DEVICE / SLOT / PAGE	Директивы sjasmplus для эмуляции переключения банков памяти 128K во время ассемблирования.	DEVICE, SLOT, PAGE	Гл15	Гл20, Гл21
DISPLAY	Директива sjasmplus, печатающая значение во время ассемблирования. Диагностика на этапе сборки.	DISPLAY	Гл20	Гл21
\$FF	Шестнадцатеричная нотация. Префикс \$ предпочтителен. #FF также принимается sjasmplus.	\$FF	Гл01	все
%10101010	Двоичная нотация.	%10101010	Гл01	все
.label	Локальная метка, ограниченная ближайшей охватывающей глобальной меткой.	.label	Гл01	все

Термин	Определение	Каноническая форма	Впервые	Также
sjasmplus	Основной ассемблер (v1.21.1). Полный набор инструкций Z80/Z80N, макросы, скрипting на Lua, DEVICE/SLOT/PAGE для банковости, INCBIN, множество форматов вывода.	«sjasmplus»	Гл01	Гл14, Гл20, Гл21, Гл23

G. Демосцена и культура

Термин	Определение	Каноническая форма	Впервые	Также
Компо	Соревнование на демопати. Категории включают демо, интро (с ограничением по размеру), музыку, графику.	«компо»	Гл13	Гл20
Демопати	Мероприятие, где показываются и оцениваются продукции демосцены. Крупные ZX-пати: Chaos Constructions, DiHalt, CAFe, Multimatograf (Россия); Revision, Forever (Западная Европа).	«демопати» или «пати»	Гл20	-

Термин	Определение	Каноническая форма	Впервые	Также
NFO / file_id.diz	Текстовые файлы, прилагаемые к релизам демо, содержащие кредиты, требования и иногда технические заметки.	«NFO» / «file_id.diz»	Гл20	-
Making-of	Послерелизная статья, документирующая процесс разработки и технические решения демо. Публикуется на Нуре или в дисковых журналах.	«making-of»	Гл12	Гл20
Часть / эффект	Визуальный раздел демо (туннель, скроллер, сфера и т.д.). Несколько частей секвенируются движком демо.	«часть» или «эффект»	Гл09	Гл12, Гл20
Скриптовый движок	Система, секвенирующая части демо и синхронизирующая их с музыкой. Два уровня: внешний скрипт (последовательность эффектов) + внутренний скрипт (вариации параметров внутри эффекта).	«скриптовый движок»	Гл09	Гл12, Гл20

Термин	Определение	Каноническая форма	Впервые	Также
kWORK	Команда Introspec'a: «сгенерировать N кадров, затем показать их независимо от генерации.» Мост между скрипtingом и асинхронной генерацией кадров.	«kWORK»	Гл09	Гл12
Запиля- тор	Сленг российской сцены для «прекалькуля- торного» демо – такого, которое предвычисляет все кадры перед воспроизведени- ем. Несёт лёгкое неодобрение (подразумевает отсутствие рендеринга в реальном времени).	«запилятор»	Гл09	-

Ключевые персоны

Имя	Роль	Главы
Dark	Кодер, X-Trade. Автор статей по программированию в Spectrum Expert. Кодер Illusion.	Гл01, Гл04, Гл05, Гл06, Гл07, Гл10
Introspec (spke)	Кодер, Life on Mars. Провёл реверс-инжиниринг Illusion. Автор статей на Hype (технические анализы, серия GO WEST, DOWN_HL). Кодер демо Eager.	Гл01–Гл12, Гл15, Гл23

Имя	Роль	Главы
n1k-o	Музыкант, Skrju. Сочинил саундтрек Eager. Разработал технику гибридных барабанов совместно с Introspec'ом.	Гл09, Гл12
diver4d	Кодер, 4D+ТВК (Triebkraft). Демо GABBA (CAFe 2019). Первопроходец рабочего процесса синхронизации через видеоредактор (Luma Fusion).	Гл09, Гл12
DenisGrachev	Кодер. Old Tower, движок GLUF, движок Ringo, Dice Legends. Техника RET-цепочки.	Гл03, Гл08
Robus	Кодер. Система многопоточности Z80, демо WAYHACK.	Гл12
psndcj (cyberjack)	Кодер, 4D+ТВК (Triebkraft). Экспертиза AY/TurboSound. Демо Break Space (эффект Magen Fractal).	Гл01, Гл07, Гл14
Screamer (sq)	Кодер, Skrju. Исследование оптимизации крупнопиксельной графики (Born Dead #05, Hype). Руководство по среде разработки.	Гл01, Гл07
Ped7g	Peter Helcmanovsky. Мейнтайнер sjasmplus, контрибьютор ZX Spectrum Next. Фидбек по знаковой арифметике и RLE.	Гл04, Гл14, Прил. F

Имя	Роль	Главы
RST7	Кодер. Оптимизация DOWN_HL с двойным счётчиком.	Гл02

Ключевые демо и продукции

Название	Автор/Группа	Год	Главы
Illusion	X-Trade (Dark)	1996	Гл01, Гл04, Гл05, Гл06, Гл07, Гл10
Eager (to live)	Introspec / Life on Mars	2015	Гл02, Гл03, Гл09, Гл10, Гл12
GABBA	diver4d / 4D+TBK	2019	Гл12
WAYHACK	Robus	-	Гл12
Old Tower	DenisGrachev	-	Гл08
Lo-Fi Motion	-	-	Гл20

Ключевые публикации

Название	Описание	Главы
Spectrum Expert (#01-#02)	Российский дисковый журнал ZX (1997-98). Учебники Dark'a по программированию.	Гл01, Гл04, Гл05, Гл06, Гл10, Гл11
Hype	Российская онлайн-платформа демосцены (hype.retroscene.org). Технические статьи, making-of'ы, дебаты.	Гл01-Гл12, Гл23
Born Dead	Российская газета демосцены. Исследование sq по крупнопиксельной графике (#05, ~2001).	Гл07
Black Crow	Российский журнал ZX-сцены. Ранняя документация по мультиколору (#05, ~2001).	Гл08

Н. Алгоритмы и сжатие

Термин	Определение	Каноническая форма	Впервые	Также
Умножение сдвигом и сложением	Классическое беззнаковое умножение 8x8. Сканирование битов множителя, сдвиг-и-сложение. 196–204 такта (T-state).	«умножение сдвигом и сложением»	Гл04	Гл05
Умножение через таблицу квадратов	$A*B = ((A+B)^2 - (A-B)^2)/4$ через таблицу подстановки. ~61 такт (T-state). Обмен 512 байт таблиц на скорость.	«умножение через таблицу квадратов»	Гл04 Гл07	Гл05, Гл07
Логарифмическое деление	$\log(A/B) = \log(A) - \log(B)$. Два поиска по таблице + вычитание. ~50–70 тактов (T-state). Низкая точность.	«логарифмическое деление»	Гл04	Гл05
Параболическая аппроксимация синуса	Приближение синуса параболой $y = 1 - 2(x/\pi)^2$. Макс. ошибка ~5,6%. 256-байтная таблица, знаковая.	«параболическая аппроксимация синуса»	Гл04	Гл05, Гл06, Гл09
Линия Брезенхэма	Шаг по главной оси с аккумулятором ошибки для шагов по второстепенной оси. ~80 тактов/пиксель наивно; ~48 с методом матрицы 8x8 Dark'a.	«линия Брезенхэма»	Гл04	-

Термин	Определение	Каноническая форма	Впервые	Также
Метод средней точки	Полностью вращать только базисные вершины; остальные получать усреднением. ~36 тактов на производную вершину против ~2 400 для полного вращения.	«метод средней точки»	Гл05	-
Арифметика с фиксированной точкой	Представление дробных значений в целочисленных регистрах. Типичный формат: 8.8 (8-бит целая часть + 8-бит дробная).	«фиксированная точка»	Гл04	Гл05, Гл18, Гл19
AABB-столкновение рекрытия	Тест осевых ограничивающих прямоугольников. Четыре сравнения: лево-право и верх-низ для двух прямоугольников. ~70-156 тактов (T-state).	«AABB»	Гл19	Гл21
Отсечение задних граней	Пропуск полигонов, обращённых назад, с помощью теста Z-компоненты нормали через векторное произведение. ~500 тактов (T-state) на грань.	«отсечение задних граней»	Гл05	-

Термин	Определение	Каноническая форма	Впервые	Также
ZX0	Формат сжатия от Einar Saukas. Отличная степень сжатия, умеренная скорость распаковки.	«ZX0»	Гл14	Гл20
LZ4	Быстрая распаковка (~34 такта/байт). Выбор для потоковой передачи в реальном времени.	«LZ4»	Гл14	-
Exomizer	Упаковщик с высокой степенью сжатия для 8-битных платформ. Медленная распаковка.	«Exomizer»	Гл14	-
MegaLZ	Формат сжатия. Хороший баланс степени/скорости.	«MegaLZ»	Гл14	-
hrust1	Формат сжатия, распространённый в российской демосцене.	«hrust1»	Гл14	Гл20

Этот глоссарий извлечён из всех 23 глав книги «Кодируя невозможное». Подробное описание любого термина см. в главе, указанной в столбце «Впервые».

Приложение А: Краткий справочник инструкций Z80

«Инструкция, которая должна занимать 7 тактов (T-state), может занять 13, если попадёт на худшую фазу цикла конкуренции.» – Глава 1

Это не полное руководство по Z80. Это справочная карточка для кодеров демосцены и программистов игр на ZX Spectrum и Agon Light 2 – инструкции, которые ты действительно используешь, тайминги, которые нужно знать наизусть, и паттерны, сберегающие такты (T-state) во внутренних циклах.

Все значения тактов (T-state) указаны для **таймингов Pentagon** (без конкуренции). Размер в байтах – длина кодировки инструкции. Столбцы флагов: **S** (знак), **Z** (нуль), **H** (полуперенос), **P/V** (чётность/переполнение), **N** (вычитание), **C** (перенос). Прочерк означает без изменений; точка означает не определено.

8-битные инструкции загрузки

Инструкция	Байты	Такты (T-state)	Флаги	Примечания
LD r, r'	1	4	—	Самая быстрая инструкция. r,r' = A,B,C,D,E,H,L
LD r, n	2	7	—	Непосредственная загрузка
LD r, (HL)	1	7	—	Чтение памяти через HL
LD (HL), r	1	7	—	Запись в память через HL
LD (HL), n	2	10	—	Непосредственное значение в память

Инструкция	Байты	Такты (T-state)	Флаги	Примечания
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	—	Абсолютный адрес
LD (nn),A	3	13	—	Абсолютный адрес
LD r,(IX+d)	3	19	—	Индексная. Дорогая - избегай во внутрен- них циклах
LD (IX+d),r	3	19	—	Индексная
LD (IX+d),n	4	23	—	Индексная непосредственная
LD A,I	2	9	SZ0P0-	P/V = IFF2
LD A,R	2	9	SZ0P0-	P/V = IFF2; R = счётчик ре- генерации

16-битные инструкции загрузки

Инструкция	Байты	Такты (T-state)	Флаги	Примечания
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 (префикс ED)
LD (nn),rr	4	20	—	rr = BC, DE, SP (префикс ED)
LD SP,HL	1	6	—	Установка указателя стека
LD SP,IX	2	10	—	

Инструкция	Байты	Такты (T-state)	Флаги	Примечания
PUSH rr	1	11	—	rr = AF, BC, DE, HL. 5,5Т на байт
POP rr	1	10	—	5Т на байт – самое быстрое чтение 2 байт
PUSH IX	2	15	—	
POP IX	2	14	—	

8-битная арифметика и логика

Инструкция	Байты	Такты (T-state)	Флаги	Примечания
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	Сложение с переносом
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	
SBC A, r	1	4	SZ.V1C	Вычитание с переносом
CP r	1	4	SZ.V1C	Сравнение (SUB без сохранения результата)
CP n	2	7	SZ.V1C	
CP (HL)	1	7	SZ.V1C	
AND r	1	4	SZ1P00	Н всегда установлен, C всегда сброшен
AND n	2	7	SZ1P00	

Инструкция	Байты	Такты (T-state)	Флаги	Примечания
OR r	1	4	SZ0P00	Сбрасывает H и C
OR n	2	7	SZ0P00	
XOR r	1	4	SZ0P00	XOR A = обнуление A за 4T/1B (против LD A, 0 = 7T/2B)
XOR n	2	7	SZ0P00	
INC r	1	4	SZ.V0-	Не влияет на перенос
DEC r	1	4	SZ.V1-	Не влияет на перенос
INC (HL)	1	11	SZ.V0-	Чтение-модификация-запись
DEC (HL)	1	11	SZ.V1-	Чтение-модификация-запись
NEG	2	8	SZ.V1C	A = 0 - A (дополнительный код, отрицание)
DAA	1	4	SZ.P-C	Коррекция BCD – редко используется в демо
CPL	1	4	-1-1-	A = NOT A (обратный код)
SCF	1	4	-0-00	Установить флаг переноса. N,H сброшены. Новое поведение на CMOS
CCF	1	4	-.0.	Инвертировать перенос. H = старый C

16-битная арифметика

Инструкция	Байты	Такты (T-state)	Флаги	Примечания
ADD HL, rr	1	11	-.?0C	rr = BC, DE, HL, SP. Влияет только на H, N, C
ADC HL, rr	2	15	SZ.V0C	Полный набор флагов
SBC HL, rr	2	15	SZ.V1C	Полный набор флагов
INC rr	1	6	—	Флаги не затрагива- ются
DEC rr	1	6	—	Флаги не затрагива- ются
ADD IX, rr	2	15	-.?0C	rr = BC, DE, IX, SP

Ключевой момент: INC rr и DEC rr **не** устанавливают флаг нуля. Нельзя использовать DEC BC / JR NZ как 16-битный счётчик цикла. Используй DEC B / JR NZ для 8-битных циклов с DJNZ, или проверяй BC явно.

Сдвиги и вращения

Инструкция	Байты	Такты (T-state)	Флаги	Примеча- ния
RLCA	1	4	-0-0C	Вращение A влево, бит 7 в перенос и бит 0
RRCA	1	4	-0-0C	Вращение A вправо, бит 0 в перенос и бит 7
RLA	1	4	-0-0C	Вращение A влево через перенос

Инструкция	Байты	Такты (T-state)	Флаги	Примечания
RRA	1	4	-0-0C	Вращение А вправо через перенос. Ключевая для циклов умножения
RLC r	2	8	SZ0P0C	СВ- префикс. Полный набор флагов
RRC r	2	8	SZ0P0C	
RL r	2	8	SZ0P0C	Вращение влево через перенос
RR r	2	8	SZ0P0C	Вращение вправо через перенос
SLA r	2	8	SZ0P0C	Арифметический сдвиг влево. Бит 0 = 0
SRA r	2	8	SZ0P0C	Арифметический сдвиг вправо. Бит 7 сохранён (расширение знака)
SRL r	2	8	SZ0P0C	Логический сдвиг вправо. Бит 7 = 0
RLC (HL)	2	15	SZ0P0C	Чтение- модификация- запись
RL (HL)	2	15	SZ0P0C	Прокрутка пиксель- ных данных влево

Инструкция	Байты	Такты (T-state)	Флаги	Примечания
RR (HL)	2	15	SZ0P0C	Прокрутка пиксельных данных вправо
SLA (HL)	2	15	SZ0P0C	
SRL (HL)	2	15	SZ0P0C	
RLD	2	18	SZ0P0-	Вращение полубайтов (HL) влево через А. Полезно для ниблевой графики
RRD	2	18	SZ0P0-	Вращение полубайтов (HL) вправо через А

Заметка для демосцены: RLA/RRA (4T, 1 байт) влияют только на перенос и биты 3,5 регистра F. СВ-префиксные версии RL r/RR r (8T, 2 байта) устанавливают все флаги. В циклах умножения версии с аккумулятором экономят половину стоимости.

Битовые операции

Инструкция	Байты	Такты (T-state)	Флаги	Примечания
BIT b, r	2	8	.Z1.0-	Проверка бита b регистра
BIT b, (HL)	2	12	.Z1.0-	Проверка бита b памяти
SET b, r	2	8	—	Установить бит b регистра

Инструкция	Байты	Такты (T-state)	Флаги	Примечания
SET b, (HL)	2	15	—	Установить бит b памяти. Используется при рисовании линий
RES b, r	2	8	—	Сбросить бит b регистра
RES b, (HL)	2	15	—	Сбросить бит b памяти

Переходы, вызовы, возврат

Инструкция	Байты	Такты (T-state)	Флаги	Примечания
JP nn	3	10	—	Абсолютный переход
JP cc,nn	3	10	—	Условный: NZ, Z, NC, C, PO, PE, P, M. Однаковая скорость при выполнении и невыполнении
JR e	2	12	—	Относительный переход (-128 до +127)
JR cc,e	2	12/7	—	cc = только NZ, Z, NC, C. 7T при невыполнении

Инструкция	Байты	Такты (T-state)	Флаги	Примечания
JP (HL)	1	4	—	Переход по адресу в HL. Самый быстрый косвенный переход
JP (IX)	2	8	—	Переход по адресу в IX
DJNZ e	2	13/8	—	Dec B, переход если NZ. 13T при выполнении, 8T при невыполнении
CALL nn	3	17	—	Сохранить PC, перейти к nn
CALL cc,nn	3	17/10	—	10T при невыполнении
RET	1	10	—	Извлечь PC. Используется для RET-цепочки диспетчеризации
RET cc	1	11/5	—	5T при невыполнении
RST p	1	11	—	Вызов по адресу \$00,\$08,\$10,\$18,\$20,\$28

Ключевые сравнения для диспетчеризации:

Метод	Такты (T-state)	Байты
CALL nn	17	3
RET (как диспетчер в RET-цепочке)	10	1
JP (HL)	4	1
JP nn	10	3

Инструкции ввода/вывода

Инструкция	Байты	Такты (T-state)	Флаги	Примечания
OUT (n),A	2	11	—	Адрес порта = (A << 8) n. Бордюр: OUT (\$FE),A
IN A,(n)	2	11	—	Адрес порта = (A << 8) n. Клавиатура: IN A,(\$FE)
OUT (C),r	2	12	—	Адрес порта = BC. Запись в регистр AY
IN r,(C)	2	12	SZ0P0-	Адрес порта = BC. Устанавливает флаги
OUTI	2	16	.Z..1.	Out (HL) впорт (C), inc HL, dec B
OTIR	2	21/16	01..1.	Повторять OUTI пока B=0. 16T на последнем
OUTD	2	16	.Z..1.	Out (HL) впорт (C), inc HL, dec B

Адреса портов AY-3-8910 на ZX Spectrum 128K:

Порт	Адрес	Назначение
Выбор регистра	\$FFFD	LD BC,\$FFFD : OUT (C),A
Запись данных	\$BFFD	LD B,\$BF : OUT (C),r
Чтение данных	\$FFFD	IN A,(C)

Типичная последовательность записи в регистр AY (24T + накладные расходы):

```

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

```

Блочные инструкции

Инструкция	Байты	Такты (T-state)	Флаги	Примечания
LDI	2	16	-0.0-	(DE) = (HL), inc HL, inc DE, dec BC. P/V = (BC != 0)
LDIR	2	21/16	-000-	Повторять LDI. 21T на байт, 16T последний байт
LDD	2	16	-0.0-	(DE) = (HL), dec HL, dec DE, dec BC
LDDR	2	21/16	-000-	Повторять LDD. 21T на байт, 16T последний байт
CPI	2	16	SZ.?1-	Сравнить A с (HL), inc HL, dec BC
CPIR	2	21/16	SZ.?1-	Повторять CPI. Останавливается при совпадении или BC=0
CPD	2	16	SZ.?1-	Сравнить A с (HL), dec HL, dec BC
CPDR	2	21/16	SZ.?1-	Повторять CPD

Стоимость LDI vs LDIR на байт:

Метод	На байт	256 байт	32 байта	Экономия
LDIR	21T (16T последний)	5 371T	672T	-
LDI-цепочка	16T	4 096T	512T	На 24% быстрее

Развёрнутая LDI-цепочка стоит 2 байта на каждую LDI (\$ED \$A0), но экономит 5T на байт – на 24% быстрее, чем LDIR. Подробнее в главе 3 об арифметике точки входа для LDI-цепочек.

Обмен и разное

Инструкция	Байты	Такты (T-state)	Флаги	Примечания
EX DE,HL	1	4	—	Обмен DE и HL. Бесплатный обмен указателей
EX AF,AF'	1	4	—	Обмен AF с теневым AF'
EXX	1	4	—	Обмен BC,DE,HL с BC',DE',HL'. 6 регистров за 4T
EX (SP),HL	1	19	—	Обмен HL с вершиной стека. Полезно для передачи параметров
EX (SP),IX	2	23	—	Обмен IX с вершиной стека

Инструкция	Байты	Такты (T-state)	Флаги	Примечания
DI	1	4	—	Запретить прерывания. Обязательно перед трюками со стеком
EI	1	4	—	Разрешить прерывания. Отложено на одну инструкцию
HALT	1	4+	—	Ожидание прерывания. Инструкция синхронизации кадра
NOP	1	4	—	Заполнение, выравнивание таймингов
IM 1	2	8	—	Режим прерываний 1 (RST \$38). Стандартный режим Spectrum
IM 2	2	8	—	Режим прерываний 2. Использует регистр I как старший байт таблицы векторов

«Быстрые» инструкции демосцены

Это самые дешёвые инструкции в каждой категории – строительные блоки каждого оптимизированного внутреннего цикла.

Самый быстрый межрегистровый перенос

LD r, r' – **4T, 1 байт.** Минимальная стоимость любой инструкции Z80. Включает LD A,A (фактически NOP, не влияющий на флаги).

Самый быстрый способ обнулить регистр

XOR A – 4T, 1 байт. Устанавливает A в ноль, устанавливает флаг Z, сбрасывает перенос. Сравни с LD A,0 при 7T/2 байта. Всегда используй XOR A, если не нужно сохранить флаги.

Самое быстрое чтение из памяти

LD A,(HL) – 7T, 1 байт. Минимальная стоимость любого чтения из памяти. Другие источники-указатели (LD A,(BC), LD A,(DE)) тоже 7T/1 байт, но HL – единственный указатель, поддерживающий LD r,(HL) для всех регистров.

Самая быстрая запись в память

LD (HL),r – 7T, 1 байт. Наравне с чтением. Запись по указателям BC или DE (LD (BC),A, LD (DE),A) тоже 7T/1B, но работает только с A.

Самая быстрая запись 2 байт

PUSH rr – 11T, 1 байт на 2 байта = **5,5T на байт.** Самый быстрый способ записи данных в память, но только туда, куда указывает SP (вниз). Требует DI и перехвата указателя стека. Подробнее в главе 3.

Самое быстрое чтение 2 байт

POP rr – 10T, 1 байт на 2 байта = **5T на байт.** Даже быстрее, чем PUSH для чтения. Используй с SP, указывающим на таблицу данных, для сверхбыстрого поиска.

Самое быстрое блочное копирование

Метод	На байт	Примечания
Пара PUSH/POP	5,25T	Запись: 5,5T, Чтение: 5T. Но нужен перехват SP
LDI (развёрнутая цепочка)	16T	Без настройки на каждый байт. На 24% быстрее LDIR
LDIR	21T	Одна инструкция, но медленная на байт

Метод	На байт	Примечания
LD (HL), r + INC HL	13T	Тело ручного цикла (без счётчика)
LD (HL), r + INC L	11T	Работает только в пределах 256-байтной страницы

Самый быстрый ввод/вывод

OUT (n), A – **11T, 2 байта**. Для фиксированных адресов портов (бординг и т.д.). Для переменных портов (AY) OUT (C), r при 12T/2 байта – единственный вариант.

Самый быстрый обмен указателей

EX DE, HL – **4T, 1 байт**. Мгновенный обмен содержимого DE и HL. Никакой другой обмен регистров не стоит так дёшево. EXX тоже 4T/1 байт, но меняет все три пары одновременно.

Самый быстрый условный цикл

DJNZ e – **13T при выполнении, 8T при невыполнении, 2 байта**. Декрементирует В и переходит. Сравни с DEC B / JR NZ, e при $4+12 = 16T/3$ байта. DJNZ экономит 3T и 1 байт за итерацию.

Самый быстрый косвенный переход

JR (HL) – **4T, 1 байт**. Переход по адресу в HL. Несмотря на вводящую в заблуждение мнемонику, эта инструкция НЕ читает из памяти по (HL) – она загружает в РС значение HL. Незаменима для таблиц переходов и вычисляемых goto.

Недокументированные инструкции

Эти инструкции отсутствуют в официальной документации Zilog, но надёжно работают на всех кристаллах Z80 (NMOS и CMOS), всех клонах ZX Spectrum и на eZ80. Они широко используются в коде демосцены и поддерживаются sjasmplus.

IXH, IXL, IYH, IYL (полурегистры индексных регистров)

Регистры IX и IY можно разделить на 8-битные половины, обращаясь к ним через DD/FD-префикс к обычным инструкциям H/L:

Инструкция	Байты	Такты (T-state)	Примечания
LD A, IXH	2	8	Чтение старшего байта IX
LD A, IXL	2	8	Чтение младшего байта IX
LD IXH, A	2	8	Запись старшего байта IX

Инструкция	Байты	Такты (T-state)	Примечания
LD IXL,n	3	11	Непосредственное значение в младший байт IX
ADD A,IXL	2	8	Арифметика с половинами IX
INC IXH	2	8	Инкремент старшего байта IX
DEC IXL	2	8	Декремент младшего байта IX

Применение в демосцене: Два дополнительных 8-битных регистра для счётчиков, аккумуляторов или малых значений, не затрагивая основной регистровый файл. Особенно полезно, когда BC/DE HL все заняты как указатели. Стоимость: на 4T больше, чем эквивалентная операция с основным регистром.

Синтаксис sjasmplus: IXH, IXL, IYH, IYL (также принимает NX, LX, HY, LY).

SLL r (логический сдвиг влево)

Инструкция	Байты	Такты (T-state)	Примечания
SLL r	2	8	Сдвиг влево, бит 0 устанавливается в 1 (а не в 0)
SLL (HL)	2	15	То же для памяти

SLL сдвигает влево и устанавливает бит 0 в 1 (в отличие от SLA, которая устанавливает бит 0 в 0). Опкод: CB 30+г. Иногда полезна для построения битовых паттернов.

Синтаксис sjasmplus: SLL или SLI или SL1.

OUT (C),0

Инструкция	Байты	Такты (T-state)	Примечания
OUT (C),0	2	12	Вывод нуля в порт BC

Опкод ED 71. Выводит ноль впорт, адресуемый BC. На CMOS Z80 (включая eZ80) вместо этого выводит \$FF. **Не портируется на Agon Light 2.** На NMOS Z80 (все настоящие Спектрумы) надёжно выводит \$00.

Синтаксис sjasmplus: OUT (C),0.

Недокументированные битовые операции с CB-префиксом над (IX+d)

Инструкции вида SET b,(IX+d),r одновременно выполняют битовую операцию над памятью по (IX+d) и копируют результат в регистр r. Это 4-байтные инструкции (DD CB dd op), занимающие 23T. Иногда полезны, но редко критически важны.

Шпаргалка по влиянию на флаги

Знание того, какие инструкции устанавливают какие флаги, позволяет избежать избыточных СР или AND A – типичного источника потерянных тактов (T-state).

Инструкции, устанавливающие все арифметические флаги (S, Z, H, P/V, N, C)

- ADD A, r/n/(HL) – P/V = переполнение
- ADC A, r/n/(HL) – P/V = переполнение
- SUB r/n/(HL) – P/V = переполнение
- SBC A, r/n/(HL) – P/V = переполнение
- CP r/n/(HL) – Те же флаги, что у SUB, но А не изменяется
- NEG – P/V = переполнение
- ADC HL, rr – P/V = переполнение
- SBC HL, rr – P/V = переполнение

Инструкции, устанавливающие Z и S (но НЕ перенос)

- INC r / DEC r – С без изменений. **Нельзя проверять перенос после INC/DEC.**
- INC (HL) / DEC (HL) – То же
- AND r/n/(HL) – С всегда 0, Н всегда 1
- OR r/n/(HL) – С всегда 0, Н всегда 0
- XOR r/n/(HL) – С всегда 0, Н всегда 0
- IN r, (C) – С без изменений
- BIT b, r/(HL) – Z = инверсия проверяемого бита, С без изменений
- Все сдвиги/вращения с CB-префиксом – Полный набор флагов, включая С

Инструкции, устанавливающие ТОЛЬКО флаги, связанные с переносом

- ADD HL, rr – Только Н и С (S, Z, P/V без изменений)
- RLCA / RRCA / RLA / RRA – Только С, Н=0, N=0 (S, Z, P/V без изменений)
- SCF – С=1, Н=0, N=0
- CCF – С инвертирован, Н = старый С, N=0

Инструкции, НЕ устанавливающие флаги

- LD (все варианты)
- INC rr / DEC rr (16-битные inc/dec)
- PUSH / POP (кроме POP AF, восстанавливающей флаги)
- EX / EXX
- DI / EI / HALT / NOP
- JP / JR / DJNZ / CALL / RET / RST
- OUT (n),A / IN A,(n) (версии без CB)

Практические трюки

Проверка А на ноль без СР 0:

and a ; 4T same effect, but also sets H

Проверка переноса после 16-битного INC/DEC: Невозможно. INC rr/DEC rr не устанавливают флаги. Чтобы проверить, достиг ли 16-битный регистр нуля:

or c ; 4T Z set if BC = 0

Пропуск СР после SUB: Если ты уже выполнил SUB r, флаги установлены – не добавляй после него СР или OR A.

INC/DEC сохраняют перенос: Используй INC r/DEC r между операциями многобайтной арифметики, не разрушая цепочку переноса.

Архитектура регистров

Основной набор регистров

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")

Специальные регистры

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)

Теневые регистры

A' F'	Swapped with EX AF,AF'
B' C'	\
D' E'	Swapped all three with EXX
H' L'	/

EXX меняет BC/DE HL на BC'/DE'/HL' за **4T**. Это даёт тебе шесть дополнительных 8-битных регистров (или три дополнительные 16-битные пары) практически бесплатно. Типичное использование: хранить указатели в теневом наборе и переключать по мере необходимости.

Внимание: Обработчик прерываний ROM Spectrum (IM1) использует IY (он должен указывать на системные переменные по адресу \$5C3A или на безопасную область памяти). Теневые регистры BC'/DE'/HL' и AF' сохраняются обработчиком ROM и безопасны для использования при разрешённых прерываниях. Если твой код использует IY для других целей, сначала запрети прерывания (DI) или переключись на IM2 с собственным обработчиком.

Привязка регистровых пар к инструкциям

Пара	Используется в	Примечания
BC	DJNZ (только B), OUT (C), IN r, (C), блочные инструкции (счётчик)	B = счётчик цикла, C = младший байт порта
DE	EX DE,HL, LDI/LDIR (приёмник), LD (DE),A	Указатель-приёмник для блочных операций
HL	Почти всё: LD r, (HL), ADD HL,rr, JP (HL), PUSH/POP, LDI (источник)	Универсальный указатель
AF	PUSH AF/POP AF, EX AF,AF'	A = аккумулятор, F = флаги
SP	PUSH/POP, LD SP,HL, EX (SP),HL	Перехват для трюков с данными

Типичные последовательности инструкций

Вычисление адреса пикселя (экранный адрес из Y,X)

Преобразование экранных координат в адрес видеопамяти ZX Spectrum. Вход: B = Y (0-191), C = X (0-255). Выход: HL = адрес байта в экране, A = битовая маска.

```
; Input: B = Y (0-191), C = X (0-255)
; Output: HL = byte address, A = pixel bit position
; Cost: ~107 T-states
;
pixel_addr:
    ld a, b          ; 4T   A = Y
    and $07          ; 7T   scanline within char cell (Y:2-0)
    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 right 3
    rra              ; 4T   /
    and $18          ; 7T   mask Y:4-3 (third bits)
    or h              ; 4T   H = 010 TT SSS
    ld h, a          ; 4T
    ld a, b          ; 4T   A = Y again
    and $38          ; 7T   mask Y:5-3 (character row within third)
    rlca             ; 4T   \ rotate left 2 to get
    rlca             ; 4T   / Y:5-3 in bits 7-5
    ld l, a          ; 4T   L = RRR 00000 (partial)
    ld a, c          ; 4T   A = X
    rra              ; 4T   \
    rra              ; 4T   | X / 8
    rra              ; 4T   /
```

```

and $1F          ; 7T  mask to 5-bit column
or l             ; 4T  combine row and column
ld l, a          ; 4T  L = RRR CCCCC

```

DOWN_HL: сдвиг на одну строку пикселей вниз

Самый используемый графический примитив на Spectrum. Типичный случай (внутри знакоряда) стоит всего 20T.

```

; 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)

```

Беззнаковое умножение 8x8 (сдвигом и сложением)

Из Dark / X-Trade, Spectrum Expert #01 (1997). Используется в матрицах вращения и преобразованиях координат.

```

; 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

```

Запись в регистр AY

Стандартная последовательность для записи в звуковой чип AY-3-8910 на ZX Spectrum 128K.

```
; 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-битное сравнение (HL с DE)

В Z80 нет прямого 16-битного сравнения. Используй SBC и восстановление.

```
; 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
```

Заливка экрана через стек

Самый быстрый способ залить экран паттерном. Подробнее в главе 3.

```
; 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
```

Быстрый перебор строк пикселей (раздельные счётчики)

Из анализа DOWN_HL Introspec'a (Hype, 2020). Устраняет все условные переходы из внутреннего цикла. Общая стоимость для 192 строк: 2 343T против 5 922T для наивных вызовов DOWN_HL.

```
; 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
```

Таблица быстрых сравнений стоимости

Для решений во внутренних циклах эти сравнения важнее всего:

Операция	Медленный способ	Быстрый способ	Экономия
Обнулить A	LD A, 0 (7T, 2B)	XOR A (4T, 1B)	3T, 1B
Проверить A=0	CP 0 (7T, 2B)	OR A (4T, 1B)	3T, 1B
Скопировать 1 байт (HL)→(DE)	LD A, (HL)+LD (DE), A+INC HL+INC DE (26T, 4B)	LDI (16T, 2B)	10T, 2B на байт
Скопировать N байт	LDI R (21T/байт)	N x LDI (16T/байт)	На 24% быстрее, стоит 2N байт кода

Операция	Медленный способ	Быстрый способ	Экономия
Заполнить 2 байта	LD (HL), A+INC HL x2 (26T)	PUSH rr (11T)	На 58% быстрее, нужен перехват SP
8-битный цикл	DEC B+JR NZ (16T, 3B)	DJNZ (13T, 2B)	3T, 1B за итерацию
Косвенный вызов	CALL nn (17T, 3B)	RET через списокрендеринга (10T, 1B)	7T, 2B за диспетчеризацию
Обмен регистров	LD A, H+LD H, D+LD D, A (12T, 3B)	EX DE, HL (4T, 1B)	8T, 2B
Сохранить 6 регистров	3 x PUSH (33T, 3B)	EXX (4T, 1B)	29T, 2B

Справочник по размеру кодировки инструкций

Для sizecoding и оценки плотности кода:

Префикс	Инструкции	Доп. байты	Доп. такты (T-state)
Нет	Большинство 8-битных операций, LD, ADD, INC, PUSH, POP, JP, JR	0	0
CB	Битовые операции, сдвиги, вращения над регистрами	+1	обычно +4
ED	Блочные операции, 16-битные ADC/SBC, IN/OUT (C), LD rr,(nn)	+1	различно
DD	Операции с индексацией через IX	+1	+4 до +8
FD	Операции с индексацией через IY	+1	+4 до +8
DD CB	Битовые/сдвиговые/вращательные над (IX+d)	+2	+8 до +12

Совет для sizecoding: Избегай инструкций с индексацией через IX/IY, когда возможно. LD A, (IX+5) – 3 байта/19T. LD L, 5 / LD A, (HL) – 3 байта/11T, если H уже содержит страницу. Индексные регистры удобны, но дороги.

Источники: 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); Глава 1 (тестовая обвязка); Глава 3 (паттерны инструментария); Глава 4 (умножение, деление)

Приложение В: Генерация таблиц синусов и тригонометрические таблицы

«Косинус – это просто синус со сдвигом на четверть периода.» –
Заповеди Raider'a

Каждый эффект демо, который изгибается – вращение, плазма, скроллинг, туннели – нуждается в таблице синусов. На Z80 ты заранее вычисляешь значения в таблицу подстановки и индексируешь по углу. Вопрос в том, как хранить и обращаться к этой таблице максимально эффективно.

Это приложение сравнивает восемь подходов к хранению таблицы синусов, от очевидного (256-байтная таблица) до экзотического (2-битное кодирование вторыми разностями). Данные получены из `verify/sine_compare.py`, который ты можешь запустить, чтобы воспроизвести все числа.

Стандартный формат

Таблица синусов для демосцены содержит **256 записей**, индексированных по углу:

Индекс	Угол
0	0°
64	90°
128	180°
192	270°
256 (обращается в 0)	360°

Каждая запись – **знаковый байт** (-128 до +127), представляющий значения от -1.0 до приблизительно +1.0. Степень двойки в качестве периода означает, что индекс угла обличивается естественным образом при 8-битном переполнении, а косинус становится синусом при прибавлении 64:

```
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
```

Это ключевое правило Raider'a: загрузи Н один раз старшим байтом таблицы, затем L - это угол, и он свободно оборачивается.

Сравнение подходов

#	Подход	Дан- ные	Код	Итого	ОЗУ	Макс. ошибка	СКО
1	Полная таблица (256 байт)	256	0	256	0	0	0,00
2	Четверть- волновая таблица	65	21	86	0	0	0,00
3	Параболи- ческая аппрокси- мация	0	38	38	0	8	4,51
4	Четверть- волна + вторые разности	18	45	63	64	0	0,00
5	Аппрокси- мация Бхаскары I	0	~60	~60	0	1	0,49
5b	Бхаскара I + битовая карта коррекции	1	~80	~81	0	0	0,00
6	Четверть- волна + упакован- ные 4-битные разности	33	43	76	64	0	0,00
7	Полные вторые разности, 2-битная упаковка	66	30	96	256	0	0,00

Подходы делятся на три категории:

- **На основе таблиц** (ОЗУ не требуется): полная таблица, четвертьволновая таблица
 - **На основе генерации** (нужен буфер ОЗУ при старте): подходы с разностями и вторыми разностями
 - **Приближённые** (вообще без таблицы): параболический, Бхаскара I
 - **Приближённые + точная коррекция:** Бхаскара I с битовой картой коррекции
-

Подход 1: Полная 256-байтная таблица

Самый простой и самый быстрый. Предвычисляешь все 256 значений и встраиваешь как данные.

```
ld   h, high(sin_table)
ld   l, a
ld   a, (hl)
```

Стоимость: 256 байт ПЗУ. **Скорость:** 18 тактов (T-state) на поиск. **Когда использовать:** Всегда, если ты не занимаешься sizecoding. На 48K Spectrum с ~40K свободными 256 байт – ничто. Это выбор по умолчанию.

Подход 2: Четвертьволновая таблица

Синусоида обладает четырёхкратной симметрией. Первый квадрант (0° до 90° , индексы 0 до 64) содержит всю информацию:

- **Второй квадрант** (65-128): зеркало первого квадранта. $\sin(128 - i) = \sin(i)$.
- **Третий квадрант** (129-192): отрицание первого квадранта. $\sin(128 + i) = -\sin(i)$.
- **Четвёртый квадрант** (193-255): отрицательное зеркало. $\sin(256 - i) = -\sin(i)$.

Храни только 65 байт (индексы от 0 до 64 включительно), затем восстанавливай:

```
; 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

```

Стоимость: 65 байт данных + ~21 байт кода = **86 байт всего.** **Скорость:** ~50-70 тактов (T-state) на поиск (зависит от квадранта). **Ошибка:** Ноль. **Когда использовать:** Демо с ограничением по размеру (256-байтные, 512-байтные интро), где нужны точные значения, но нельзя позволить себе 256 байт.

Подход 3: Параболическая аппроксимация (метод Dark'a)

Из Dark / X-Trade, *Spectrum Expert #01* (1997). Идея: половина периода косинуса похожа на параболу. Приближение $y \sim 1 - 2(x/\pi)^2$ близко совпадает. В целочисленных терминах каждая половина периода генерируется как кусочно-квадратичная функция.

Чистый код, ноль данных. Цикл генерации требует умножения 8x8 и некоторой аккумуляторной логики – приблизительно 38 байт.

Ошибка ограничена: **максимальная абсолютная ошибка = 8** (из диапазона в 256 шагов), или примерно **6,3%** от полной шкалы. СКО = 4,51.

Вот где парабола расходится с истинным синусом (первый квадрант):

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

Парабола стабильно «опережает» – она растёт быстрее вблизи нуля и более плоская вблизи пика. Максимальное расхождение: **8 единиц при индексе 17** (примерно 24°).

Когда использовать: Экстремальный sizecoding (64-байтные интро, компактные загрузчики). Плазмы, простые скроллеры и эффекты колебания, где глаз не различает точную кривизну от приблизительной. Не подходит для гладкого вращения или точного проволочного 3D.

Подход 4: Кодирование вторыми разностями (глубокий трюк)

Это математически самый интересный подход и, безусловно, самое компактное точное представление.

Ключевое наблюдение

Вторая производная $\sin(x)$ равна $-\sin(x)$. При 8-битной целочисленной точности, квантованной в знаковые байты, вторая конечная разность таблицы синусов обладает замечательным свойством: **каждое значение равно ровно -1, 0 или +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, ...]
```

Три значения. Два бита на запись. Это не приближение – это точно. Математическая причина: $d^2(\sin)/dx^2$ – гладкая функция малой амплитуды, и при 256 записях на период с 8-битной амплитудой дискретная вторая производная никогда не превышает $+/ -1$.

Полная таблица через 2-битные вторые разности

Хранишь: начальное значение (1 байт), начальную разность (1 байт), затем 254 вторые разности, упакованные по 2 бита (64 байта). **Итого: 66 байт данных + ~30 байт декодирования = 96 байт**. Нужно 256 байт ОЗУ для декодирования.

Четвертьволна через 2-битные вторые разности

Комбинируй с четвертьволновой симметрией: храни только первые 64 вторые разности. **Итого: 18 байт данных + ~45 байт декодирования = 63 байта**. Нужно 64 байта ОЗУ.

Это **наименьшее точное представление**: 63 байта всего для идеальной 256-записной таблицы синусов.

```
; 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

```

Декодирование выполняется один раз при запуске. После этого используй подпрограмму четвертьволнового поиска из подхода 2 на декодированном буфере.

Когда использовать: Демо с ограничением по размеру (128-байтные, 256-байтные интро), где нужны точные значения, есть 64 байта ОЗУ и краткая фаза инициализации. Цикл декодирования выполняется менее чем за 2 000 тактов (T-state) – незаметно.

Врезка: Почему не 1 бит на разность?

Интуитивное возражение: четвертьволна синуса (0° до 90°) монотонно возрастает. Первые разности d_1 всегда неотрицательны. В непрерывной математике вторая производная синуса в первом квадранте всегда отрицательна (кривая вогнута). Значит, d_2 должно быть ≤ 0 , то есть нам нужно только $\{-1, 0\}$ – один бит на запись.

Интуиция верна для непрерывного синуса, но неверна для квантованного целочисленного. При 8-битной точности округление создаёт случайные скачки вверх в d_1 :

```

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)

```

Таких записей $+1 - 12$ из 63. Если их подавить (ограничить d_1 монотонным невозрастанием), ошибки *накапливаются*: к индексу 64 пик достигает лишь 108 вместо 127 – максимальная ошибка 19, хуже параболической аппроксимации. Эти коррекции $+1$ несут именно ту информацию, которая нужна для попадания в правильные целочисленные значения. Их нельзя отбросить.

Код переменной длины с префиксом ($0 \rightarrow 1$ бит, $+/1 \rightarrow 2$ бита) экономит 4 байта данных по сравнению с фиксированным 2-битным кодированием, но стоит ~ 15 дополнительных байт логики декодирования на Z80. Чистый проигрыш. Фиксированное 2-битное кодирование – практический оптимум.

Врезка: Почему парабола + коррекция не помогает

Ещё одна интуитивная идея: сгенерировать параболическую аппроксимацию (38 байт кода, макс. ошибка 8), затем сохранить небольшую

таблицу коррекции для доведения до точных значений. Коррекции лежат в диапазоне от -8 до +8, так что они должны хорошо сжиматься.

Коррекции *действительно* хорошо сжимаются – их первые разности ровно {-1, 0, +1}, упаковывающиеся в 2 бита на запись. Но это не совпадение. Парабола – это квадратичная функция с постоянной второй производной. Значит:

- $d2(\sin)$ из {-1, 0, +1} – вторая производная синуса при целочисленной точности
- $d2(para)$ из {-1, 0, +1} – вторая производная параболы (почти постоянна)
- $d1(correction) = d1(\sin) - d1(para)$ из {-1, 0, +1} – **та же энтропия**

Разности коррекции имеют *точно ту же структуру*, что и прямые вторые разности синуса. Но параболический путь добавляет 38 байт кода генерации плюс ~20 байт для применения коррекций. Итого: ~96 байт против 63 байт для прямого кодирования вторыми разностями.

Парабола убирает гладкую (низкочастотную) компоненту синуса – но 2-битное кодирование вторыми разностями уже идеально справляется с гладкими данными. Параболе нечего добавить, что вторые разности уже не охватывают. Код генерации – чистые накладные расходы.

Подход 5: Аппроксимация Бхаскары I (VII век)

Самая удивительная позиция в нашем сравнении пришла от индийского математика VII века Бхаскары I. Его рациональная аппроксимация синуса, опубликованная около 629 г. н.э., достигает **максимальной ошибки всего в 1 единицу** при 8-битной точности – радикально лучше, чем параболическая аппроксимация (макс. ошибка 8) и почти точно.

Формула

Для угла x в радианах (от 0 до π):

$$\sin(x) \sim 16x(\pi - x) / (5\pi^2 - 4x(\pi - x))$$

В нашей целочисленной области (угол 0-64 для первого квадранта, амплитуда 0-127):

$$\begin{aligned}\sin(i) &\sim 127 * 16i(64 - i) / (5 * 64^2 - 4 * i(64 - i)) \\ &= 127 * 16i(64 - i) / (20480 - 4i(64 - i))\end{aligned}$$

Формула – это отношение двух квадратичных функций. На Z80 это требует умножения 8x8 и 16-битного деления – подпрограммы, которые во многих демо уже есть для 3D-проекции или текстурного маппинга.

Точность

По 65 записям первого квадранта аппроксимация Бхаскары I совпадает с точным целочисленным синусом везде, кроме **8 позиций** (из 65), где отклонение ровно +/-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

Только 8 позиций различаются, все ровно на +/-1. Ошибки распределены: 2 записи, где Бхаскара завышает (вблизи начала), 6, где занижает (вблизи пика). Всего восемь коррекций, которые кодируются одним байтом – битовой картой.

Реализация на Z80

Для реализации нужно: - Подпрограмма умножения 8x8->16 (~20 байт, вероятно, уже есть) - Подпрограмма деления 16/16->16 (~30 байт, вероятно, уже есть) - Обёртка Бхаскары (~25 байт) - Логика четвертьволнового свёртывания (~15 байт, общая с подходом 2)

Если в твоём демо уже есть подпрограммы умножения и деления, маргинальная стоимость – примерно **25 байт** для функции синуса с максимальной ошибкой 1.

Если подпрограммы нужно писать с нуля, суммарно получается примерно **60 байт** кода при нуле байт данных. Это конкурентно с подходом вторых разностей (63 байта), но не требует ни буфера ОЗУ, ни фазы декодирования при старте. Компромисс: 1 единица ошибки против идеальной точности.

Бхаскара I + битовая карта коррекции (точно)

Чтобы устраниТЬ последнюю единицу ошибки, сохрани 8 позиций коррекции в виде битовой карты. Поскольку коррекции симметричны (первые 4 нуждаются в +1, последние 4 – в -1), хватит одного байта:

```
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
```

Итого: ~80 байт кода + 1 байт данных = **~81 байт**, ноль ОЗУ, ноль инициализации, точные значения. Дороже вторых разностей (63B), но позволяет обойтись без буфера ОЗУ и инициализации при старте.

Когда использовать Бхаскару I

- **У тебя уже есть подпрограммы умножения/деления:** ~25 байт сверху, макс. ошибка 1. Трудно побить.
- **Нет ОЗУ для буфера декодирования:** В отличие от вторых разностей, Бхаскара вычисляет на лету.
- **Нужна генерация в реальном времени:** Каждое значение вычисляется независимо – нет последовательной зависимости, так что можно вычислить sin(любой угол) без предварительного декодирования таблицы.
- **Ошибка +/-1 допустима:** Для скроллеров, плазм и большинства визуальных эффектов разница между максимальной ошибкой 1 и максимальной ошибкой 0 буквально невидима.

Историческая справка: Формула Бхаскары I предшествует европейским тригонометрическим таблицам почти на тысячелетие. То, что рациональная аппроксимация VII века достигает максимальной ошибки 1 на 8-битном процессоре 1980-х – красивое совпадение математической элегантности и инженерных ограничений. Формула была опубликована в *Махабхаскарии* (629 г. н.э.), комментарии к астрономическим методам Арьябхаты.

Практические рекомендации

Каждый подход на основе генерации создаёт таблицу подстановки при старте. После этого стоимость в рантайме одинакова: LD H, high(table) / LD L, A / LD A, (HL) = **18 тактов (T-state)** для 256-байтной таблицы, или подпрограмма четвертьволнового свёртывания при **50-70 тактах (T-state)** для 64-байтного буфера. Столбец «стоимость ПЗУ» ниже – это то, что важно для sizecoding – общее число байт, которое твой подход занимает в бинарнике.

Сценарий	Подход	Стоимость ПЗУ	ОЗУ	Инициализация	Поиск	Ошибка
Обычное демо / игра	Полная 256-байтная таблица	256B	0	нет	18 T	точно
512-байтное интро	Четвертьволновая таблица	86B	0	нет	50-70 T	точно
256-байтное интро	Четвертьволна + вторые разности	63B	64B	~2K T	50-70 T	точно
Есть умножение/деление	Бхаскара I (генерация в LUT)	~25B доп. (генерация в LUT)	256B	~80K T	18 T	макс. +/-1

Сценарий	Подход	Стоимость ПЗУ	ОЗУ	Инициализация	Поиск	Ошибка
128-байтное интро	Параболическая (генерация в LUT)	38B	256B	~10K T	18 T	макс. +/-8

Дерево решений

- Есть 256 байт в запасе?** Используй полную таблицу. Не усложняй. LD L,A / LD A,(HL) при 18 тактах (T-state) не побить.
- Ограничение по размеру, но нужна точность?** Четвертьволновая таблица при 86 байтах. ОЗУ не нужно, фазы инициализации нет. Поиск 50-70 тактов (T-state) (логика свёртывания).
- Экстремальное ограничение по размеру, нужны точные значения?** Четвертьволна + декодирование вторых разностей при 63 байтах. Декодируй один раз при старте в 64-байтный четвертьволновый буфер, затем используй ту же подпрограмму свёртывания.
- Уже есть умножение/деление?** Бхаскара I при ~25 байтах сверху. Сгенерируй полную 256-байтную LUT при старте, затем наслаждайся поиском за 18 тактов (T-state) с макс. ошибкой 1.
- Экстремальное ограничение по размеру, приближённость допустима?** Параболическая при 38 байтах, ноль данных. Генерируй в 256-байтную LUT при старте. Макс. ошибка 8, годится для плазм и колебаний.

Что не работает

- Парабола + таблица коррекции** (123 байта, точно): хуже, чем просто использовать четвертьволновую таблицу (86 байт). Накладные расходы на вычисление параболы и поиск коррекции перечёркивают цель.
- Разности + RLE** (100-219 байт): разности синуса меняются плавно, а не повторяются сериями. RLE рассчитан на данные с длинными постоянными сериями – синус не той формы.
- Полная таблица с дельта-кодированием** (152-271 байт): использует больше суммарных байт, чем сырья 256-байтная таблица. Дельта-кодирование помогает только когда разности значительно меньше исходных значений; разности синуса ограничены +/-4, но их всё равно 256.

Заповеди Raider'a

В комментариях Нуре к анализу *Illusion* от Introspec'a опытный кодер Raider сконденсировал десятилетия коллективной мудрости в неформальные «заповеди» проектирования таблиц синусов:

1. **256 записей на полный период.** Индекс угла оборачивается при 8-битном переполнении. Модульная арифметика не нужна.
2. **Знаковые байты: -128 до +127.** Соответствует знаковой арифметике Z80.
3. **Выровняй таблицу по странице.** Размести её на 256-байтной границе, чтобы H был константой. LD H,high(table) один раз, затем LD L,angle / LD A,(HL) навсегда.
4. **Косинус = синус + 64.** Одна инструкция ADD A,64.
5. **Синус от (angle + 128) = -синус(angle).** NEG инвертирует знак. Используй для фазовых сдвигов.
6. **Не вычисляй синус в рантайме,** если не занимаешься sizecoding. Поиск по таблице всегда быстрее.
7. **Амплитуду держи степенью двойки** (64, 127, 128), чтобы умножение было сдвигом.
8. **Четвертьволновая симметрия** экономит 75% хранилища, когда каждый байт на счету.
9. **Проверяй на границах.** Индекс 0 должен быть ровно 0. Индекс 64 – максимальное положительное значение (+127). Индекс 128 – ровно 0. Индекс 192 – максимальное отрицательное значение (-128 или -127, в зависимости от соглашения).

Эти правила отражают десятилетия опыта. Следуй им, и твои таблицы синусов будут быстрыми, компактными и корректными.

Справочник: Полная 256-байтная таблица

Для удобства приведена стандартная таблица синусов (256 записей, знаковая, период = 256, амплитуда +/-127):

```
; 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
```

540ПРИЛОЖЕНИЕ В: ГЕНЕРАЦИЯ ТАБЛИЦ СИНУСОВ И ТРИГОНОМЕТРИЧЕСКИЕ ТАБЛ

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, -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

Скопируй, вставь, собери, используй.

Источники: Dark / X-Trade “Programming Algorithms” (Spectrum Expert #01, 1997) – параболическая аппроксимация; Бхаскара I, *Махабхаскария* (629 г. н.э.) – рациональная аппроксимация; Raider (комментарии Нура, 2017) – принципы проектирования таблиц синусов; verify/sine_compare.py – сравнительный анализ

Приложение С: Краткий справочник по сжатию

«Вопрос не в том, сжимать ли – а в том, какой упаковщик использовать и когда.» – Глава 14

Это приложение – отрывная справочная карточка по сжатию данных на ZX Spectrum. Глава 14 покрывает теорию, бенчмарк-данные и обоснование каждой рекомендации. Это приложение сводит всё к таблицам подстановки и правилам принятия решений, которые можно закрепить над монитором.

Все числа взяты из бенчмарка Introspec'a 2017 года ("Data Compression for Modern Z80 Coding," Нуре), если не указано иное. Тестовый корпус составлял 1 233 995 байт смешанных данных: академические бенчмарки Calgary/Canterbury, 30 графических экранов ZX Spectrum, 24 музыкальных файла и разнообразные данные демо.

Сравнительная таблица упаковщиков

Упаковщик	Автор	Сжато (байт)	Степень сжатия	Размер распаковщика	Скорость (Т/байт)	Обратная	Применения
Exomizer2	Magnus Lind	596 161	48,3%	~170 байт	~250	Да	Лучшая степень сжатия. Медленная распаковка.

Упаковщик	Автор	Сжато (байт)	Степень сжатия	Размер распаковщика	Скорость (Т/байт)	Обратная	Примечания
ApLib	Joergen Ibsen	606 833	49,2%	~199 байт	~105	Нет	Хороший универсал.
Hrust 1	Alone Coder	613 602	49,7%	~150 байт	~120	Да	Стековый распаковщик с перемещением. Популярен в российской сцене.
PuCrunch	Pasi Ojala	616 855	50,0%	~200 байт	~140	Нет	Изначально для C64.
Pletter 5	XL2S	635 797	51,5%	~120 байт	~69	Нет	Быстрый + приятная степень сжатия.

Упаковщик	Автор	Сжато (байт)	Степень сжатия	Размер распаковщика	Скорость (Т/байт)	Обратная	Примечания
MegaLZ	LVD / Introspec	636 910	51,6%	92 байта (компактный)	~98 (компактный)	Нет	Опти-маль-ный пар-сер. Воз-рож-дён в 2019 с но-вы-ми рас-па-ков-щи-ка-ми.
MegaLZ fast	LVD / Introspec	636 910	51,6%	234 байта	~63	Нет	Самый быст-рый ва-ри-ант MegaLZ. Быст-рее 3x LDIR.

Упаков-щик	Ав-тор	Сжато (байт)	Степень сжа-тия	Размер рас-паковщика	Скорость (Т/байт)	Обрат-ная	При-ме-ча-ния
ZX0	Einar Saukas	~642 000*	~52%	~70 байт	~100	Да	На-след-ник ZX7. Оп-ти-маль-ный пар-сер. Со-вре-мен-ный стан-дарт.
ZX7	Einar Saukas	653 879	53,0%	69 байт	~107	Да	Кро-шеч-ный рас-па-ков-щик. Клас-си-че-ский ин-стру-мент для sizecoding.
Bitbuster	Team Bomba	~660 000*	~53,5%	90 байт	~80	Нет	Про-стой. Хо-рош для пер-вых про-ек-тов.

Упаковщик	Автор	Сжато (байт)	Степень сжатия	Размер распаковщика	Скорость (Т/байт)	Обратная	Примечания
LZ4	Yann Collet (порт на Z80)	722 522	58,6%	~100 байт	~34	Нет	Самая быстрая распаковка. Байты выровненные токены.
Hrum	Hrumer	~642 000*	~52%	~130 байт	~110	Нет	Популярен в российской среде. Объявлен устаревшим Introspec'ом.
ZX1	Einar Saukas	—	~51%	~80 байт	~90	Да	Вариант ZX0. Чуть лучше степень сжатия, чуть больше распаковщик.

Упаковщик	Автор	Сжато (байт)	Степень сжатия	Размер распаковщика	Скорость (Т/байт)	Обратная	Примечания
ZX2	Einar Saukas	—	~50%	~100 байт	~85	Да	Использован в RED REDUX 256b intro (2025). Лучшая степень сжатия для ZXn.

* Приблизительно. ZX0, Bitbuster и Hrum не входили в оригинальный бенчмарк 2017 года; значения оценены по независимым тестам на аналогичных корпусах.

Как читать таблицу:

- **Степень сжатия** = сжатый размер / исходный размер. Меньше – лучше.
- **Скорость** = такты (T-state) на выходной байт при распаковке. Меньше – быстрее.
- **Размер распаковщика** = байты кода Z80, необходимые для подпрограммы распаковки. Меньше – лучше для размерных интро.
- **Обратная** = поддерживает распаковку с конца к началу, что позволяет распаковывать на месте, когда источник и приёмник перекрываются.

Дерево решений: какой упаковщик?

Следуй сверху вниз. Выбери первую ветку, подходящую к твоей ситуации.

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)

```

Степень сжатия типичных данных ZX Spectrum

Как хорошо сжимаются различные типы данных и трюки для улучшения степени сжатия.

Тип данных	Исходный размер	Типичная степень ZX0	Типичная степень Exomizer	Примечания
Пиксели экрана (\$4000-\$5FFF)	6 144 байта	40-60%	35-55%	Зависит от сложности изображения. Чёрный фон сжимается хорошо.

Тип данных	Исходный размер	Типичная степень ZX0	Типичная степень Exomizer	Примечания
Атрибуты (\$5800-\$5AFF)	768 байт	30-50%	25-45%	Часто очень повторяющиеся. Однотонные области сжимаются почти вничто.
Полный экран (пиксели + атрибуты)	6 912 байт	40-58%	35-52%	Сжимаемый пиксели и атрибуты раздельно для улучшения степени на 5-10%.
Таблицы синусов/косинусов	256 байт	60-75%	55-70%	Плавные кривые сжимаются хорошо. Рассмотрри генерацию вместо сжатия (Приложение В).

Тип данных	Исходный размер	Типичная степень ZX0	Типичная степень Exomizer	Примечания
Тайловые данные (тайлы 8x8)	разное	35-55%	30-50%	Упорядочи тайлы по подобию для лучшей степени.
Спрайтовые данные	разное	45-65%	40-60%	Байты масок ухудшают степень. Храни маски отдельно.
Музыка РТЗ	разное	40-55%	35-50%	Паттерны повторяются. Пустые строки сжимаются хорошо.
Дампы регистров AY	разное	30-50%	25-45%	Сильно повторяются между кадрами. Сначала дельта-кодируй.

Тип данных	Исходный размер	Типичная степень ZX0	Типичная степень Exomizer	Примечания
Таблицы подстановки (произвольные)	разное	50–80%	45–75%	Случайно выглядящие данные сжимаются плохо. Отсортируй, если можно.
Данные шрифтов (96 символов x 8 байт)	768 байт	55–70%	50–65%	Много нулевых байт (выносные элементы, тонкие штрихи).

Трюки перед сжатием

Эти техники улучшают степень сжатия, реструктурируя данные перед подачей в упаковщик.

Раздели пиксели и атрибуты. Полный 6 912-байтный экран, хранимый одним блоком, заставляет упаковщик обрабатывать переход от пиксельных данных к атрибутам на байте 6 144. Сжимай 6 144-байтный блок пикселей и 768-байтный блок атрибутов раздельно. Блок атрибутов, будучи сильно повторяющимся, часто сжимается до менее 200 байт.

Дельта-кодируй кадры анимации. Храни первый кадр полностью. Для каждого последующего кадра храни только байты, отличающиеся от предыдущего, как пары (смещение, значение). Применяй LZ-сжатие к дельта-потоку. psndcj скжал 122 кадра (843 264 байта исходных) до 10 512 байт, используя эту технику в Break Space.

Переупорядочь данные для локальности. Тайловые карты в порядке «строка за строкой» могут сжиматься лучше, если переупорядочить так, чтобы похожие тайлы шли рядом. Сортируй кадры спрайтов по визуальному подобию. Группируй повторяющиеся подпаттерны вместе.

Храни константы отдельно. Если блок данных содержит повторяющийся

заголовок или завершитель (например, метаданные тайлов), вынеси его и храни один раз. Сжимай только переменную часть.

Чередуй плоскости. Для мультиколорных или маскированных спрайтов хранение всех байтов масок вместе и всех байтов пикселей вместе часто сжимается лучше, чем чередование маска-пиксель-маска-пиксель построчно.

Минимальный RLE-распаковщик

Простейший полезный упаковщик. Всего 12 байт кода. Подходит для 256-байтных интро или данных с длинными сериями одинаковых байт. Подробное обсуждение в главе 14.

```
; 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: 12 bytes of code
; Speed: ~26 T-states per output byte (within long runs)
;         + 46T overhead per [count, value] pair
```

Инструмент кодирования (однострочник на Python для простого RLE):

```
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
```

Этот наивный RLE раздувает данные без серий (худший случай: 2 байта на 1 байт входных данных). Для смешанных данных используй RLE с байтом-маркером: специальный байт сигнализирует о серии, а все остальные – литералы. Или просто используй ZX0.

Трюк с транспонированием. RLE выигрывает радикально при хранении данных по столбцам. Если у тебя есть блок атрибутов 32×24 , где каждая строка различается, но столбцы часто одинаковы, транспонирование данных (сначала все значения столбца 0, потом столбца 1 и т.д.) создаёт длинные серии, которые RLE сжимает хорошо. Компромисс: Z80 должен обратно транспонировать данные после распаковки, что стоит дополнительного прохода (~13 тактов (T-state) на байт для простого вложенного цикла копирования). Посчитай общую стоимость (код распаковщика + код обратного транспонирования + сжатые данные) и сравни с ZX0 (распаковщик + сжатые данные, без трансформации), чтобы понять, что выгоднее для твоих конкретных данных.

Стандартный распаковщик ZX0 (Z80)

Полный стандартный прямой распаковщик от Einar Saukas. Приблизительно 70 байт. Это версия, которую ты будешь использовать в большинстве проектов.

```
; (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

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

```

Использование:

```

ld      de, $4000      ; destination (e.g., screen)
call    dzx0_standard  ; decompress

```

Обратный вариант. ZX0 также предоставляет обратный распаковщик (dzx0_standard_back), который читает сжатые данные с конца к началу и записывает выходные данные с конца к началу. Это позволяет распаковку на месте: размести сжатые данные в конце буфера-приёмника и распаковывай назад, чтобы выходные данные перезаписывали сжатые только после того, как те были прочитаны. Незаменимо, когда ОЗУ мало.

Паттерны интеграции

Паттерн 1: Распаковка на экран при старте

Самый распространённый сценарий. Загрузить сжатый загрузочный экран и показать его.

```
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"
```

Паттерн 2: Распаковка в буфер между эффектами

Распаковать данные следующего эффекта во временный буфер, пока текущий эффект ещё работает, или во время затемнения.

```
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
```

Паттерн 3: Потоковая распаковка во время воспроизведения

Для эффектов реального времени, которым нужен непрерывный поток данных. LZ4 – единственный практичный выбор.

```
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
```

При ~34 Т/байт LZ4 распаковывает 2 048 байт за 69 632 такта (T-state) – укладываясь в один кадр (69 888 тактов на 48К). Это впритык. Используй распаковку во время бордюра или двойную буферизацию для надёжности.

Паттерн 4: Сжатые данные с переключением банков (128К)

Храни сжатые данные в нескольких 16КБ банках. Распаковывай из текущего подключённого банка, затем переключай банк, когда данные кончаются.

```

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

```

Для больших демо с множеством сжатых ресурсов поддерживай таблицу кортежей (банк, смещение, приёмник) и проходи по ним при загрузке.

Конвейер сборки: от ресурса к бинарнику

Шаг сжатия должен быть в твоём Makefile, а не в голове.

Source asset	Converter	Compressor	Assembler
(PNG)	--> (png2scr)	--> (zx0)	--> (sjasmplus) --> .tap
(WAV)	--> (pt3tools)	--> (zx0)	--> (incbin)
(TMX)	--> (tmx2bin)	--> (exomizer)	

Правила Makefile:

```

%.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=$@

```

Установка инструментов:

Инструмент	Исходный код	Установка
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 или brew install lz4
LZ4	github.com/lz4/lz4	
MegaLZ	github.com/AntonioCerra/megalz	Следуй; ссылки смотри в статье Introspec'a на Hype

Быстрые формулы

Байт за кадр при 50fps с распаковщиком X:

`bytes_per_frame = 69,888 / speed_t_per_byte`

Упаковщик	T/байт	Байт/кадр (48K)	Байт/кадр (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

(Кадр 128K Pentagon = 71 680 тактов)

Память, сэкономленная сжатием N экранов:

`saved = N * 6912 * (1 - ratio)`

Пример: 8 загрузочных экранов с Exomizer при степени сжатия 48,3% экономят $8 * 6912 * 0,517 = 28\ 575$ байт – почти два полных 16КБ банка.

См. также

- **Глава 14:** Полное обсуждение теории сжатия, бенчмарка Introspec'a, внутренностей ZX0 и конвейера дельта + LZ.
- **Приложение В:** Генерация таблиц синусов – когда таблицы достаточно малы, рассмотри генерацию вместо сжатия.
- **Приложение А:** Справочник инструкций Z80 – LDIR, PUSH/POP и другие инструкции, используемые в распаковщиках.

Источники: 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)

Приложение D: Настройка среды разработки

«Тебе нужно пять вещей: редактор, ассемблер, эмулятор, отладчик и Makefile. Всё остальное — optionalno.» – Глава 1

Это приложение проведёт тебя через настройку полноценной среды разработки для Z80 с нуля. К концу ты сможешь компилировать все примеры на ассемблере из этой книги, запускать их в эмуляторе и отлаживать с помощью точек останова и просмотра регистров. Инструкции охватывают macOS, Linux и Windows.

Если ты уже выполнил настройку из Главы 1, большая часть у тебя уже на месте. Это приложение добавляет подробности, описывает альтернативные конфигурации и служит единым справочником, к которому можно вернуться при настройке новой машины.

1. Ассемблер: sjasmplus

Каждый пример кода в этой книге написан для **sjasmplus** — открытого макроассемблера Z80/Z80N от z00m128. Он поддерживает полный набор инструкций Z80, включая все индексные режимы IX/IY, макросы, Lua-скрипting, множество выходных форматов и выражения, которые делают написание демосценового кода практическим.

Установка из исходников

Самый надёжный способ получить sjasmplus — собрать его из исходников. Это гарантирует известную версию и избавляет от платформенных проблем с пакетными менеджерами.

```
cd sjasmplus  
make
```

На macOS тебе понадобятся инструменты командной строки Xcode (`xcode-select --install`). На Linux — `g++` и `make` (устанавливаются через пакетный менеджер). На Windows используй MinGW или WSL.

После сборки скопирай бинарный файл `sjasmplus` куда-нибудь в PATH:

```
sudo cp sjasmplus /usr/local/bin/
```

```
# Verify
sjasmplus --version
```

Ты должен увидеть версию 1.20.x или новее. Эта книга разрабатывалась и тестировалась с v1.21.1.

Привязка версии

Репозиторий книги закрепляет sjasmplus как git-подмодуль в tools/sjasmplus/. Если ты клонируешь репозиторий с --recursive, то получишь именно ту версию, которая использовалась для компиляции каждого примера:

```
cd antique-toy/tools/sjasmplus
make
```

Это самый безопасный подход. Поведение ассемблера может меняться между версиями — выражение, работающее в 1.21, может интерпретироваться иначе в 1.22.

Ключевые флаги

Флаг	Назначение	Пример
--nologo	Подавить стартовый баннер	sjasmplus --nologo main.a80
--raw=FILE	Вывести сырой бинарник (без заголовка)	sjasmplus --raw=output.bin main.a80
--sym=FILE	Записать файл символов (для отладчиков)	sjasmplus --sym=output.sym main.a80
--fullpath	Показывать полные пути файлов в сообщениях об ошибках	Полезно с problem matcher в VS Code
--msg=war	Подавить информационные сообщения, показывать только предупреждения и ошибки	Более чистый вывод сборки
--syntax=abf	Включить все синтаксические возможности (A как псевдоним аккумулятора, скобки для косвенной адресации, полный набор «фейковых» инструкций)	Рекомендуется для начинающих; позволяет писать add a, b наряду с add b

Типичная команда сборки для примера из главы:

Расширение файла

Все файлы с исходным кодом Z80 в этой книге используют расширение .a80. Это соглашение, а не требование — sjasmplus не обращает внимания на расширения. Мы используем .a80, потому что это расширение распознаётся расширением Z80 Macro Assembler для VS Code и отличает наш исходный код от других диалектов ассемблера.

Шестнадцатеричная запись

В книге для шестнадцатеричных значений используется запись \$FF. sjasmplus также принимает #FF и 0FFh, но \$FF — стандарт для всей книги. Отдельный символ \$ обозначает текущий адрес счётчика команд и используется в конструкциях вроде jr \$ (бесконечный цикл) или dw \$ + 4.

2. Редактор: VS Code

Подойдёт любой текстовый редактор. Мы рекомендуем **Visual Studio Code** благодаря расширениям для Z80 и встроенному терминалу. Весь рабочий процесс — редактирование, сборка, отладка — происходит в одном окне.

Необходимые расширения

Установи их из магазина расширений VS Code (Ctrl+Shift+X):

Расширение	Автор	Что делает
Z80 Macro Assembler	mborik (mborik.z80-macroasm)	Подсветка синтаксиса, автодополнение, разрешение символов для Z80. Понимает синтаксис sjasmplus, включая макросы и локальные метки.
Z80 Assembly Meter	Nestor Sancho	Показывает количество байтов и стоимость в тактах выделенных инструкций в статусной строке. Выдели блок — мгновенно увидишь его общую стоимость. Незаменимо для подсчёта тактов.
DeZog	Maziac	Отладчик Z80. Подключается к эмуляторам или к собственному встроенному симулятору. Точки останова, наблюдение за регистрами, просмотр памяти. См. раздел 4.

Задача сборки

Настрой задачу сборки, чтобы Ctrl+Shift+B компилировал текущий файл. Создай .vscode/tasks.json в корне проекта:

```
"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
        }
    }
}
]
}

```

`problemMatcher` парсит вывод ошибок sjasmplus, так что при клике на ошибку в терминале курсор перейдёт на проблемную строку. Флаг `--fullpath` обеспечивает абсолютные пути файлов, которые нужны VS Code для корректного разрешения.

Рекомендуемые настройки

Добавь эти настройки в `.vscode/settings.json` рабочего пространства для комфорtnого редактирования Z80:

```

"editor.tabSize": 8,
"editor.insertSpaces": false,
"editor.rulers": [80],
"files.associations": {
    "*.a80": "z80-macroasm"
}
}

```

Размер табуляции 8 с настоящими табами соответствует традиционному ассемблерному соглашению, при котором мнемоники и операнды выравниваются по фиксированным колонкам.

3. Эмуляторы

Тебе нужен эмулятор для запуска скомпилированного кода. Разные эмуляторы служат разным целям.

ZEsarUX — полнофункциональная отладка

ZEsarUX от cerebellio — самый функциональный открытый эмулятор ZX Spectrum. Он поддерживает полный спектр моделей Spectrum (48K, 128K, +2, +3, Pentagon, Scorpion, ZX-Uno, ZX Spectrum Next), имеет встроенный отладчик и интегрируется с DeZog для отладки в VS Code.

Установка:

- macOS: `brew install zesarux`
- Linux: собери из исходников или используй AppImage с <https://github.com/chernandezba/zesarux>
- Windows: скачай установщик с сайта ZEsarUX

Почему ZEsarUX для этой книги: Большинство примеров в главах рассчитаны на ZX Spectrum 128K. ZEsarUX эмулирует переключение банков памяти 128K, звук AY, TurboSound (два чипа AY) и паттерны спорной памяти различных моделей. Его встроенный отладчик показывает регистры, память и дизассемблер без необходимости использовать VS Code. А интеграция с DeZog обеспечивает полноценную отладку в VS Code, когда это нужно.

Быстрый запуск:

```
zesarux --machine 128k --nosplash output.sna

# Run a .tap file
zesarux --machine 128k --nosplash output.tap
```

Fuse — лёгкий и точный

Fuse (the Free Unix Spectrum Emulator) от Philip Kendall — лёгкий, потактово точный и доступный на всех платформах. Лучший выбор для быстрого тестирования, когда полноценный отладчик не нужен.

Установка:

- macOS: `brew install fuse-emulator`
- Linux: `apt install fuse-emulator-sdl` (Debian/Ubuntu) или `dnf install fuse-emulator` (Fedora)
- Windows: скачай с сайта Fuse

Быстрый запуск:

```
fuse --machine pentagon output.sna

# Run as 128K Spectrum
fuse --machine 128 output.tap
```

Fuse особенно хорош для тестирования кода, чувствительного ко времени, потому что его потактовая точность хорошо проверена. Если твоя тестовая связь для полос бордюра (Глава 1) показывает разные результаты в Fuse и другом эмуляторе — доверяй Fuse.

Unreal Speccy — Windows, ориентация на Pentagon

Если ты разрабатываешь преимущественно под Windows и ориентируешься на тайминги Pentagon, **Unreal Speccy** — отличный выбор. У него встроенный отладчик с картой памяти, точками останова и мониторингом регистров AY. Он точно эмулирует аппаратуру Pentagon и Scorpion.

Скачай с <http://dlcorp.nedopc.com/viewforum.php?f=27> или поищи «Unreal Speccy Portable».

Для Agon Light 2

Agon Light 2 использует процессор eZ80 и другую аппаратную архитектуру. Глава 22 подробно описывает разработку для Agon. Для эмуляции **Fab Agon Emulator** обеспечивает программную симуляцию аппаратуры Agon (eZ80 + ESP32 VDP). Он доступен по адресу <https://github.com/tomm/fab-agon-emulator> и работает на macOS, Linux и Windows.

Какой эмулятор выбрать?

Ситуация	Рекомендуемый эмулятор
Повседневная разработка, запуск примеров	Fuse (быстрый запуск, точный)
Отладка с точками останова и наблюдением за регистрами	ZEsarUX + DeZog
Разработка музыки AY/TurboSound	ZEsarUX (лучшая эмуляция AY)
Проверка таймингов Pentagon	Fuse или Unreal Speccy
Разработка для Agon Light 2	Fab Agon Emulator
Быстрая проверка на Windows	Unreal Speccy

4. Отладчик: DeZog

DeZog от Maziac — расширение для VS Code, которое превращает редактор в отладчик Z80. Он подключается к ZEsarUX, CSpect или собственному встроенному симулятору Z80 и обеспечивает отладку, к которой привыкли современные разработчики: точки останова, пошаговое выполнение, наблюдение за регистрами, просмотр памяти, окно дизассемблера и стек вызовов.

Глава 23 рассматривает DeZog в контексте разработки с помощью ИИ. Этот раздел описывает практическую настройку.

Установка

1. Открой VS Code.
2. Перейди в расширения (Ctrl+Shift+X).
3. Найди «DeZog» от Maziac.
4. Нажми «Install».

Подключение к ZEsarUX

DeZog общается с ZEsarUX через сокетное соединение. Сначала запусти ZEsarUX с включённым сервером ZRCP (ZEsarUX Remote Control Protocol):

Затем создай `.vscode/launch.json` в своём проекте:

```
"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"
    ]
  }
]
```

Секция `sjasmplus` указывает на файл `.sld` (Source Level Debug), который `sjasmplus` генерирует с флагом `--sld=FILE`. Это даёт DeZog отладку на уровне исходного кода — точки останова на строках исходника, а не только на адресах.

Чтобы сгенерировать файл `.sld`, добавь флаг в команду сборки:

Использование встроенного симулятора

Для быстрой отладки без запуска внешнего эмулятора DeZog включает встроенный симулятор Z80. Измени `launch.json` на:

```
"type": "dezog",
"request": "launch",
```

```

"name": "DeZog Internal Simulator",
"remoteType": "zsim",
"zsim": {
    "machine": "spectrum",
    "memoryModel": "ZX128K"
},
"sjasmplus": [
{
    "path": "build/output.sld"
}
],
"topOfStack": "$FF00"
}

```

Встроенный симулятор запускается быстрее и не требует установки ZEsarUX. У него нет точной эмуляции спорной памяти, поэтому не используй его для отладки, критичной ко времени — но для логической отладки (правильно ли моя подпрограмма умножения выдаёт результат?) он идеален.

Ключевые возможности DeZog

Точки останова. Кликни в поле рядом со строкой исходного кода, чтобы установить точку останова. Выполнение приостанавливается, когда счётчик команд Z80 достигает этого адреса. Также можно установить условные точки останова (например, остановка при A == \$FF).

Наблюдение за регистрами. Панель переменных показывает все регистры Z80: AF, BC, DE, HL, IX, IY, SP, PC и альтернативный набор (AF', BC', DE', HL'). Отдельные флаги (C, Z, S, P/V, H, N) выведены отдельно для удобства чтения.

Просмотр памяти. Панель памяти показывает шестнадцатеричный дамп любого диапазона адресов. Можно ввести адрес и увидеть, что там находится. Незаменимо для проверки таблиц подстановки, содержимого экранной памяти и состояния стека.

Окно дизассемблера. Даже без отладки на уровне исходного кода DeZog дизассемблирует код вокруг текущего PC. Полезно для понимания того, как самомодифицирующийся код выглядит во время выполнения.

Стек вызовов. DeZog отслеживает пары CALL/RET и показывает стек вызовов. Это работает для обычного кода. Самомодифицирующийся код и RET-цепочки (Глава 3) сбьют трекер стека — это ожидаемо.

5. Сборка примеров из книги

Клонирование репозитория

```
cd antique-toy
```

Флаг --recursive подтягивает подмодуль sjasmplus из tools/sjasmplus/. Если ты уже клонировал без него:

Предварительные требования

Тебе нужны `make` и компилятор C++ (для сборки `sjasmpls` из подмодуля). На большинстве систем они уже установлены:

- macOS: `xcode-select --install`
- Debian/Ubuntu: `sudo apt install build-essential`
- Fedora: `sudo dnf install make gcc-c++`
- Windows: используй WSL или установи MinGW и GNU Make

Команды сборки

`Makefile` проекта делает всё. Весь скомпилированный вывод помещается в `build/`, который указан в `gitignore`.

Команда	Что делает
<code>make</code>	Скомпилировать все примеры из глав с помощью <code>sjasmpls</code>
<code>make test</code>	Собрать все примеры, вывести результат (pass/fail) для каждого
<code>make ch01</code>	Скомпилировать только примеры Главы 1
<code>make ch04</code>	Скомпилировать только примеры Главы 4
<code>make demo</code>	Собрать сопутствующее демо «Antique Toy»
<code>make clean</code>	Удалить все артефакты сборки

Успешный запуск `make test` выглядит так:

```
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
```

Если какой-либо пример не собирается, вывод покажет FAIL с именем файла. Запусти проблемный файл вручную через `sjasmpls`, чтобы увидеть подробную ошибку:

Запуск примера

После компиляции загрузи выходной бинарник в эмулятор. Конкретный способ зависит от формата вывода:

```
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
```

Большинство примеров из глав используют ORG \$8000 и производят сырье бинарники. Для их запуска либо:

1. Используй обёртку .tap (Makefile генерирует их, если исходный код содержит соответствующие директивы), либо
 2. Загрузи бинарник по адресу \$8000 в отладчике эмулятора и установи РС на \$8000.
-

6. Структура проекта для собственного кода

Когда ты начнёшь писать собственный код Z80 помимо примеров из книги, вот рекомендуемая структура каталогов:

```
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
```

Минимальный Makefile

```
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)
  $(SJASMPPLUS) $(SJASM_FLAGS) --raw=@ --sym=$(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)
```

Ключевые моменты:

- Главный файл исходного кода использует директивы INCLUDE для подключения других файлов. sjasmplus разрешает пути включений относительно каталога исходного файла.
- Флаг --sym генерирует файл символов для справки. Флаг --sld генерирует данные отладки на уровне исходного кода для DeZog.
- Перечисли все включаемые файлы как зависимости, чтобы make перекомпилировал при изменении любого файла.

Соглашение об включениях

В твоём main.a80:

```
; --- 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 обрабатывает файлы INCLUDE встраиванием, как если бы текст был вставлен в этой точке. Организуй включения по порядку: сначала библиотечные подпрограммы, затем эффекты, затем данные (поскольку данные обычно размещаются в конце бинарника).

7. Альтернативные инструменты

В этой книге используется sjasmplus, потому что это самый функциональный ассемблер Z80 для демосцены и разработки игр. Но в сообществе ты можешь встретить и другие инструменты.

Другие ассемблеры

Ассемблер	Примечания
z80asm	Часть кросс-компиляторного набора z88dk для С. Хорош, если ты смешиваешь С и ассемблер. Другие синтаксические соглашения.
RASM	От Roudoudou. Быстрый, поддерживает CPC и Spectrum. Популярен в сцене Amstrad CPC.

Ассемблер	Примечания
pasmo	Простой, переносимый, ограниченная функциональность. Подходит для маленьких автономных программ, но лишён макросов и продвинутых возможностей, необходимых для крупных проектов.

Примеры из книги используют специфичные для sjasmplus возможности (локальные метки с ., отрицательные значения DB, шестнадцатеричный префикс \$), которые могут не работать без изменений в других ассемблерах. Если ты хочешь портировать пример на другой ассемблер, изменения обычно минимальны: синтаксис меток, шестнадцатеричная нотация и имена директив.

Другие расширения VS Code

Расширение	Примечания
SjASMPlus Syntax	Альтернативная подсветка синтаксиса, настроенная специально для sjasmplus. Попробуй, если z80-macroasm неправильно подсвечивает какую-либо специфичную для sjasmplus возможность.
Z80 Debugger (Spectron)	Более старый отладчик Z80 для VS Code. DeZog в основном его заменил.

CSpect — эмулятор для Next

Если у тебя есть ZX Spectrum Next или ты ориентируешься на специфичные для Next возможности (copper, layer 2, DMA, турборежим 28 МГц), **CSpect** от Mike Dailly — эталонный эмулятор. DeZog подключается к CSpect так же, как к ZEsarUX. CSpect работает только под Windows, но запускается через Wine на macOS и Linux.

SpectrumAnalyzer

Браузерный инструмент, визуализирующий раскладку экранной памяти ZX Spectrum, конфликты атрибутов и тайминги. Полезен для понимания обсуждения чересстрочной раскладки экрана в Главе 2. Доступен по адресу <https://shiru.untergrund.net/spectrumanalyzer.html> (или поищи «ZX Spectrum screen analyzer»).

8. Устранение неполадок

«**sjasmplus: command not found**»

Бинарный файл отсутствует в PATH. Либо скопируй его в /usr/local/bin/ (macOS/Linux), либо добавь его каталог в PATH, либо задай переменную SJASMPPLUS при запуске make:

Ошибки компиляции в примерах из книги

Во-первых, убедись, что ты используешь версию sjasmplus из подмодуля (tools/sjasmplus/). Более новые или старые версии могут вести себя иначе. Во-вторых, проверь, что ты ассемблируешь файл из правильного каталога — sjasmplus разрешает пути INCLUDE относительно исходного файла, а не рабочего каталога.

DeZog не может подключиться к ZEsarUX

1. Убедись, что ZEsarUX запущен с --enable-remoteprotocol.
2. Проверь, что номер порта в launch.json совпадает с аргументом --remoteprotocol-port.
3. На macOS может потребоваться разрешить ZEsarUX через файрвол (Системные настройки > Конфиденциальность и безопасность).
4. Попробуй перезапустить ZEsarUX перед запуском сессии DeZog.

Эмулятор показывает мусор на экране

Если ты загрузил сырой бинарник и видишь мусор, наиболее вероятная причина — неправильный адрес загрузки. Примеры из книги используют ORG \$8000. Убедись, что эмулятор загружает бинарник именно по этому адресу, а не по \$0000 или другому адресу по умолчанию. Использование вывода в формате .sna или .tap (который содержит информацию об адресе) устраняет эту проблему.

Выходной файл сборки пуст или нулевого размера

Проверь, что в исходном файле есть директива ORG и хотя бы одна инструкция. sjasmplus с --raw= создаёт бинарник от первого выданного байта до последнего. Если ничего не выдано (например, файл содержит только ORG \$8000 без кода), выходной файл будет пуст.

Справочник инструментов

Инструмент	Назначение	Официальный URL	Упоминание в книге
sjasmplus	Макроассемблер Z80/Z80N	https://github.com/z00m128/sjasmplus	примеры
VS Code	Редактор и IDE	https://code.visualstudio.com/	Глава 1

Инструмент	Назначение	Официальный URL	Упоминание в книге
Z80 Macro Assembler	Расширение VS Code для синтаксиса	Marketplace: mborik.z80-macroasm	Глава 1
Z80 Assembly Meter	Отображение подсчёта тактов	Marketplace: Nestor Sancho	Глава 1
DeZog	Отладчик Z80 для VS Code	Marketplace: Maziac / https://github.com/maziac/DeZog	Глава 23
ZEsarUX	Полнофункциональный эмулятор Spectrum	https://github.com/cheptarev/ZesarUX	Глава 1, Глава 23
Fuse	Лёгкий эмулятор Spectrum	https://fuse-emulator.sourceforge.net	Глава 1
Unreal Speccy	Эмулятор с фокусом на Pentagon	http://dlcorp.nedopc.com	Глава 1
CSpect	Эмулятор ZX Spectrum Next	https://dailly.blogspot.com/2014/09/zx-spectrum-next.html	Глава 22 (возможности Next)
Fab Agon Emulator ZX0	Эмулятор Agon Light 2	https://github.com/tomfarr/fab-agon-emulator	Глава 22
Exomizer	Оптимальный LZ-компрессор	https://github.com/einafaukasz/ZX0	Глава 14, Приложение C
Vortex Tracker II	Компрессор с лучшей степенью сжатия	https://github.com/bitmantra/vortextracker	Глава 14, Приложение C
	Трекер музыки для AY	https://github.com/ivanfrantsov/vortextracker	Глава 14, Приложение C

См. также

- **Глава 1:** Первый практикум — настройка VS Code, sjasmplus и тестовой обвязки.
- **Глава 22:** Настройка платформы Agon Light 2, тулчейн eZ80, Fab Agon Emulator.
- **Глава 23:** Интеграция DeZog с рабочими процессами с помощью ИИ.
- **Приложение А:** Справочник инструкций Z80 — инструкции, которые ты будешь отлаживать.
- **Приложение G:** Справочник регистров AY — пригодится при отладке звукового кода в ZEsarUX.

Приложение Е: Краткий справочник по eZ80

«Тот же набор инструкций, совершенно другая машина.» – Глава 22

Это приложение — справочная карточка для программистов Z80, впервые подступающих к eZ80. Оно охватывает архитектурные различия, систему режимов, новые инструкции и особенности Agon Light 2, которые нужно учитывать при портировании. Это не исчерпывающее руководство по eZ80 — это подмножество, важное для работы в Главе 22.

Если ты уже знаешь Z80 (а если дочитал до сюда — знаешь), eZ80 покажется знакомым. Регистры называются так же, инструкции имеют те же мнемоники, флаги работают одинаково. Но три вещи отличаются: адреса стали 24-битными, регистры могут быть 24-битными, и есть система режимов, которая управляет активной шириной. Всё остальное следует из этого.

1. Обзор архитектуры

eZ80 — это расширенный Z80 от Zilog, разработанный для встраиваемых систем, которым нужно больше 64 КБ адресного пространства. Это строгое надмножество Z80 — каждый опкод Z80 допустим на eZ80 с идентичным поведением. Расширения добавляют 24-битную адресацию, 24-битную ширину регистров и несколько новых инструкций.

Характеристика	Z80	eZ80
Ширина адреса	16-бит (64 КБ)	24-бит (16 МБ)
Ширина регистров	16-бит (HL, BC, DE, SP, IX, IY)	16-бит или 24-бит (зависит от режима)
Размер стекового кадра на PUSH/CALL	2 байта	2 байта (режим Z80) или 3 байта (режим ADL)
Аппаратное умножение	Нет	MLT rr (8x8 без знака)
Префиксы инструкций	CB, DD, ED, FD	Те же, плюс префиксы суффиксов режима

Характеристика	Z80	eZ80
Регистр MBASE	Н/Д	Предоставляет старшие 8 бит адреса в режиме Z80
Новые инструкции	Н/Д	LEA, PEA, MLT, TST, TSTIO, SLP, IN0/OUT0, STMIX/RSMIX

Ключевая ментальная модель: в **режиме ADL** (Address Data Long) eZ80 ведёт себя как Z80 с 24-битными регистрами и 24-битными адресами. В **Z80-совместимом режиме** он ведёт себя как стандартный Z80, с 16-битными регистрами, а регистр MBASE предоставляет недостающие старшие 8 бит адреса.

2. Система режимов

У eZ80 есть два рабочих режима, управляющих шириной регистров и генерацией адресов. Понимание этих режимов — самая важная концепция для любого Z80-программиста, изучающего eZ80.

Два режима

Z80-совместимый режим. Регистры 16-битные. Адреса 16-битные, старшие 8 бит предоставляет MBASE. LD HL,\$4000 загружает 16-битное значение. PUSH HL помещает в стек 2 байта. Код ведёт себя точно так же, как на стандартном Z80.

Режим ADL (Address Data Long). Регистры 24-битные. Адреса 24-битные. LD HL,\$040000 загружает 24-битное значение. PUSH HL помещает в стек 3 байта. Это родной режим eZ80 и режим по умолчанию на Agon Light 2.

Суффиксы режима

Отдельные инструкции могут переопределять текущий режим с помощью суффиксных префиксов:

Суффикс	Значение	Ширина регистров	Ширина адреса
.SIS	Short Immediate, Short	16-бит	16-бит
.LIS	Long Immediate, Short	24-бит	16-бит
.SIL	Short Immediate, Long	16-бит	24-бит
.LIL	Long Immediate, Long	24-бит	24-бит

Первая буква (S/L) управляет шириной регистров для данной инструкции. Третья буква (S/L) управляет шириной адреса. В режиме ADL .SIS заставляет инструкцию работать как стандартный Z80. В режиме Z80 .LIL заставляет инструкцию работать в полном 24-битном режиме.

Вызовы и переходы с переключением режима

Вызовы и переходы могут переключать режим процессора в точке назначения:

Инструкция	Текущий режим	Целевой режим	Размер адреса возврата
CALL.IS nn	ADL	Z80	3 байта (конвенция ADL)
CALL.IL nn	Z80	ADL	3 байта (длинный)
JP.SIS nn	любой	Z80	Н/Д (нет адреса возврата)
JP.LIL nn	любой	ADL	Н/Д (нет адреса возврата)
RST.LIL \$08	Z80	ADL	3 байта (длинный)

Суффикс .IS означает «Instruction Short» — целевой код работает в режиме Z80. .IL означает «Instruction Long» — целевой код работает в режиме ADL.

Практическое правило

Оставайся в режиме ADL. MOS загружает Agon в режиме ADL. Вызовы MOS API ожидают режим ADL. Команды VDP отправляются через подпрограммы MOS, которые предполагают 24-битные стековые кадры. Если ты переключишься в режим Z80 и вызовешь MOS, несоответствие ширины стекового кадра разрушит стек и приведёт к краху.

Если тебе нужны плотные 16-битные циклы (например, портирование внутреннего цикла Z80 без переписывания), используй суффикс .SIS для отдельных инструкций, а не переключай весь процессор.

Ловушка MBASE

В Z80-совместимом режиме регистр MBASE предоставляет старшие 8 бит каждого адреса памяти — включая выборку инструкций. Если ты изменишь MBASE во время выполнения в режиме Z80, следующая выборка инструкции использует новое значение MBASE. Если твой код не находится по соответствующему физическому адресу, выполнение уйдёт в мусор.

Правило: если тебе необходимо использовать режим Z80, установи MBASE один раз при запуске и не трогай его. А лучше — оставайся в режиме ADL.

3. Новые инструкции

Эти инструкции существуют на eZ80, но отсутствуют на стандартном Z80. Именно они делают eZ80 чем-то большим, чем просто Z80 с расширенными регистрами.

Арифметика и тесты

Инструкция	Байт	Тактов	Описание
MLT BC	2	6	Беззнаковое умножение 8x8: B * C -> BC
MLT DE	2	6	Беззнаковое умножение 8x8: D * E -> DE
MLT HL	2	6	Беззнаковое умножение 8x8: H * L -> HL
MLT SP	2	6	Беззнаковое умножение 8x8: SPH * SPL -> SP (редко полезно)
TST A, n	3	7	Тест: A AND n, устанавливает флаги, А не изменяется
TST A, r	2	4	Тест: A AND r, устанавливает флаги, А не изменяется
TSTIO n	3	12	Тест ввода-вывода: (C) AND n, устанавливает флаги

Вычисление адресов

Инструкция	Байт	Тактов	Описание
LEA BC, IX+d	3	4	BC = IX + смещение d со знаком
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 (прибавить смещение к 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 (прибавить смещение к IY)
PEA IX+d	3	7	Поместить (IX + d) в стек
PEA IY+d	3	7	Поместить (IY + d) в стек

LEA вычисляет эффективный адрес без обращения к памяти — это чистая регистровая арифметика. На стандартном Z80 вычисление HL = IX + 5 требует многоинструкционной последовательности (PUSH IX / POP HL / LD DE,5 / ADD HL,DE). LEA делает это одной инструкцией за 4 такта.

Ввод-вывод и системные инструкции

Инструкция	Байт	Тактов	Описание
IN0 r, (n)	3	7	Чтение из внутреннего порта ввода-вывода (8-битный адрес)
OUT0 (n), r	3	7	Запись во внутренний порт ввода-вывода (8-битный адрес)
SLP	2	-	Сон: остановить CPU до прерывания (меньше энергии, чем HALT)
STMIX	2	4	Установить флаг смешанного режима (включить чередование ADL/Z80)
RSMIX	2	4	Сбросить флаг смешанного режима

IN0/OUT0 используют 8-битный адрес порта (в отличие от стандартных IN/OUT, которые могут использовать 16-битные адреса портов через BC). Они предназначены для внутренних периферийных устройств eZ80 и редко используются в игровом коде для Agon.

4. MLT — Революционная инструкция

Из всех новых инструкций eZ80 MLT оказывает наибольшее влияние на игровой и демо-код. Она выполняет беззнаковое умножение 8x8 одной инструкцией.

На стандартном Z80 умножение 8x8 требует цикла сдвигов и сложений:

```
; 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
```

На eZ80:

```
; Cost: 6 cycles, 2 bytes
    mlt bc          ; BC = B * C. Done.
```

Шесть тактов. Два байта. Никакого цикла, никакого управления переносом, никаких временных регистров. Результат попадает в полную 16-битную регистровую пару: старший байт произведения — в В, младший — в С.

Что даёт MLT

Индексация таблицы синусов. Вычисление `base + angle * stride` для синусовых таблиц с переменным шагом сокращается с вызова подпрограммы до двух инструкций (`MLT + ADD`).

Вычисление смещения спрайта. Нахождение адреса кадра спрайта N в спрайтовом листе: `base + frame * frame_size`. С MLT это тривиально, когда `frame_size` помещается в 8 бит.

Арифметика с фиксированной точкой. Умножение двух 8-битных значений с фиксированной точкой (например, скорость * трение) становится одной инструкцией вместо цикла на 200 тактов.

Адресация тайловой карты. Вычисление `tile_base + (row * width + col)`, где `width` помещается в 8 бит: один MLT для смещения строки, один ADD для столбца.

Ограничения MLT

- **Только беззнаковое.** Для знакового умножения нужно вручную скректировать знак после MLT.
 - **Только 8x8.** Для умножения 16x16 по-прежнему нужен многошаговый алгоритм (хотя его можно строить из компонентов MLT).
 - **Результат перезаписывает оба операнда.** MLT BC уничтожает и В, и С, заменяя их 16-битным произведением. Сохрани входные значения заранее, если они тебе нужны.
-

5. Сводка ключевых различий

Характеристика	Z80 (ZX Spectrum 128K)	eZ80 (Agon Light 2)
Тактовая частота	3,5 МГц	18,432 МГц
Ширина адреса	16-бит (64 КБ видимых)	24-бит (16 МБ, 512 КБ заполнено)
Ширина регистров	16-бит	16-бит (режим Z80) или 24-бит (режим ADL)
Стек на PUSH/CALL	2 байта	2 байта (режим Z80) или 3 байта (режим ADL)
Инструкция умножения	Нет (цикл сдвигов и сложений, ~200T)	MLT rr (6 тактов)
ОЗУ	128 КБ с переключением банков (8 x 16 КБ страниц)	512 КБ непрерывное

Характеристика	Z80 (ZX Spectrum 128K)	eZ80 (Agon Light 2)
Модель доступа к ОЗУ	Переключение банков через порт \$7FFD	Плоская 24-битная адресация
Видео	ULA: прямое отображение в память по адресу \$4000	VDP (ESP32): командный протокол через UART
Звук	AY-3-8910 через порты ввода-вывода	VDP-аудио через последовательные команды
Прерывания	IM1 (RST \$38) или IM2 (таблица векторов)	IM2 с векторами, управляемыми MOS
Бюджет кадра (50 Гц)	~71 680 тактов (Pentagon)	~368 640 тактов
Спорная память ОС	Да (ULA крадёт такты у \$4000-\$7FFF) Нет (голое железо)	Нет спорной памяти MOS (Machine Operating System)
Хранение	Лента / DivMMC	SD-карта (FAT32, файловый API MOS)

6. Особенности Agon Light 2

Аппаратная часть

- **CPU:** Zilog eZ80F92, 18,432 МГц
- **ОЗУ:** 512 КБ внешнего SRAM, плоское адресное пространство
- **VDP:** ESP32-PICO-D4 с FabGL, связь с eZ80 через UART на скорости 1 152 000 бод
- **Видеорежимы:** Несколько, до 640x480x64 цвета. Большинство игр используют 320x240 или 320x200.
- **Аппаратные спрайты:** До 256, полностью управляются VDP
- **Аппаратные тайловые карты:** Прокручиваемые тайловые слои, управляются VDP
- **Аудио:** Синтез VDP — синус, прямоугольник, треугольник, пилообразный, шум. ADSR-огибающие для каждого канала.
- **Хранение:** MicroSD-карта, FAT32, доступ через файловый API MOS

Бюджет кадра

При 18,432 МГц и частоте обновления 50 Гц:

$$18,432,000 \text{ cycles/sec} / 50 \text{ frames/sec} = 368,640 \text{ cycles/frame}$$

Для сравнения — Pentagon (3,5 МГц, 50 Гц): 71 680 тактов на кадр. Agon располагает примерно **5,1-кратным** бюджетом кадра. Но поскольку многие инструкции eZ80 также выполняются за меньшее число тактов, чем их Z80-аналоги, эффективная пропускная способность для типичного кода в 5-20 раз выше.

API MOS

MOS (Machine Operating System) предоставляет системный интерфейс. Стандартная точка входа — RST \$08:

```
ld a, mos_function      ; function number in A
rst $08                 ; call MOS
; return value in A (and sometimes HL)
```

Ключевые функции API MOS:

Функция	Номер	Описание
mos_getkey	\$00	Ожидание нажатия клавиши, возвращает ASCII-код
mos_load	\$01	Загрузка файла с SD-карты в память
mos_save	\$02	Сохранение области памяти в файл на SD-карте
mos_cd	\$03	Смена текущего каталога
mos_dir	\$04	Вывод содержимого каталога
mos_del	\$05	Удаление файла
mos_ren	\$06	Переименование файла
mos_sysvars	\$08	Получение указателя на системные переменные (карта клавиатуры, состояние VDP, часы)
mos_fopen	\$0A	Открытие файла, возвращает дескриптор
mos_fclose	\$0B	Закрытие дескриптора файла
mos_fread	\$0C	Чтение байтов из файла
mos_fwrite	\$0D	Запись байтов в файл
mos_fseek	\$0E	Перемещение позиции в файле

Протокол команд VDP

Команды VDP отправляются потоком байтов через RST \$10 (MOS: вывод байта в VDP). Большинство команд начинаются с VDU 23, за которым следуют параметры, специфичные для команды:

```
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
```

VDP обрабатывает команды асинхронно. Между отправкой команды и её выполнением VDP существует задержка последовательной передачи. Для плавной анимации отправляй все обновления в начале кадра.

7. Чек-лист портирования

При портировании кода Z80 со Spectrum на eZ80 в режиме ADL на Agon пройдись по этому чек-листву:

Адреса и указатели: - Все адреса становятся 24-битными (3 байта вместо 2) - Замени DW (define word) на DL (define long) для таблиц адресов - Индексация таблиц указателей меняется с * 2 на * 3 - Убедись, что старший байт 24-битных адресов корректен (обычно \$00 или \$04)

Стековые кадры: - Каждый PUSH — 3 байта, каждый CALL помещает 3-байтный адрес возврата - Проверь, что пары PUSH/POP сбалансированы — несовпадение пары разрушает 3 байта, а не 2 - Смещения относительно стека (например, доступ к параметрам, помещённым вызывающей стороной) изменяются

Блочные операции: - LDIR / LDDR используют 24-битный BC в режиме ADL — убедись, что старший байт BC равен нулю, если твой счётчик помещается в 16 бит - Трюки с блочным копированием через PUSH/POP помещают 3 байта за PUSH, а не 2

Умножение: - Замени циклы умножения сдвигом и сложением на MLT там, где это применимо - MLT BC = B * C -> BC, MLT DE = D * E -> DE, MLT HL = H * L -> HL

Ввод-вывод и периферия: - Замени портовый ввод-вывод (OUT (C), A, IN A, (\$FE)) на вызовы API MOS/VDP - Замени прямую запись в видеобуфер (\$4000-5AFF)VDP — AY(FFFD/\$BFFD) звуковыми командами VDP

Архитектура памяти: - Удали всю логику переключения банков (запись впорт \$7FFD) — плоское адресное пространство - Удали обходные пути для спорной памяти — на Agon нет спорной памяти - Удали трюки с теневым экраном — VDP сам управляет двойной буферизацией

Паттерны кода, которые становятся ненужными: - Самомодифицирующийся код для скорости (по-прежнему работает, редко стоит усложнения) - Трюки с указателем стека для быстрой заливки экрана (нет видеобуфера для заливки) - Предсмешённые спрайтовые копии (аппаратные спрайты поддерживают субпиксельное позиционирование) - Вычисления чересстрочных адресов экрана (DOWN_HL, pixel_addr — удалай)

Паттерны кода, которые переносятся напрямую: - Циклы системы существ (просто расширь указатели) - AABB-обнаружение столкновений (8-битные сравнения, без изменений) - Арифметика с фиксированной точкой 8.8 (побайтовая, без изменений) - Конечные автоматы и таблицы переходов (расширь элементы таблицы до 24 бит) - Циклы DJNZ, поиск CPIR, ветвление по флагам (всё идентично)

См. также

- **Приложение А: Краткий справочник инструкций Z80** — полная таблица инструкций Z80 с тактами, размерами в байтах и влиянием на флаги. Всё из Приложения А также применимо к eZ80 в Z80-совместимом режиме.
- **Глава 22: Порттирование — Agon Light 2** — полное пошаговое руководство по портированию с примерами кода «до» и «после» для рендеринга, звука, ввода и игровой логики.

Источники: 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); Глава 22 этой книги

Приложение F: Варианты Z80

— Расширенные наборы инструкций

«Тот же набор инструкций, совершенно другая машина.» — Глава 22

Zilog Z80 не застыл в 1976 году. За пять десятилетий оригинальную конструкцию клонировали, расширяли, переосмысливали и — в случае ZX Spectrum Next — перестроили те самые люди, которые тридцать лет проклинали его ограничения. Это приложение обозревает основные варианты Z80 и расширения их наборов инструкций с фокусом на том, что важно для демосцены и игровых программистов. Стандартный набор инструкций Z80 рассмотрен в Приложении А; eZ80 подробно описан в Приложении Е. Это приложение даёт общую картину — как варианты соотносятся друг с другом, что каждый из них добавляет и почему.

1. Генеалогическое древо Z80

Z80 был спроектирован Федерико Фаджином и Масатоси Симой в Zilog в 1976 году как программно-совместимый наследник Intel 8080. Он стал самым распространённым 8-битным процессором в истории, оживляя всё — от деловых машин на CP/M до аркадных автоматов и целого поколения домашних компьютеров. Набор инструкций был зафиксирован при выпуске и никогда официально не расширялся Zilog — до появления eZ80, два десятилетия спустя.

Но другие его расширяли. Вот семейство в общих чертах:

Вариант	Год	Примечательная машина	Ключевое добавление
Z80 (Zilog)	1976	ZX Spectrum, MSX, Amstrad CPC	Оригинал. 158 документированных инструкций.
KP1858BM1 (СССР)	~1986	Pentagon, Scorpion	Точный клон. Никаких изменений инструкций.
NSC800 (National Semi)	1980	Различные встраиваемые системы	CMOS Z80 с шиной в стиле 8085. Новых инструкций нет.

Вариант	Год	Примечательная машина	Ключевое добавление
R800 (ASCII Corp)	1990	MSX turboR	Z80-совместимый, радикально иной конвейер. MULUB, MULUW.
eZ80 (Zilog)	2001	Agon Light 2	24-битная адресация, MLT, LEA, PEA, режим ADL.
Z80N (команда Next)	2017	ZX Spectrum Next	Список желаний демосцены: MUL, MIRROR, LDIRX, PIXELDN, циклические сдвиги.

Z80 и его клоны имеют идентичный набор инструкций. R800, eZ80 и Z80N добавили инструкции для решения конкретных проблем — но очень разных проблем, отражающих очень разные цели проектирования.

2. Z80N — Список желаний демосценера

Z80N — это процессор в ZX Spectrum Next. Его спроектировали Виктор Трукко, Фабиу Белавенту и команда Next — люди, которые десятилетиями писали код для Z80 и точно знали, где больно. Каждая новая инструкция решает конкретную, хорошо задокументированную проблему из тридцатилетнего опыта программирования Spectrum. Z80N работает на 28 МГц (8-кратная тактовая частота оригинального Spectrum) и добавляет примерно 40 новых инструкций, все закодированные в ранее неиспользуемом пространстве опкодов \$ED xx.

Лучший способ понять расширения Z80N — посмотреть, какую проблему решает каждая инструкция.

Навигация по экрану (проблема DOWN_HL)

На стандартном Spectrum вычисление экранного адреса по координатам пикселя занимает 50–60 тактов и страницу кода (см. `pixel_addr` в Приложении A). Перемещение на одну строку пикселей вниз требует печально известной подпрограммы `DOWN_HL` — условного лабиринта из INC, AND, ADD и SUB, обрабатывающего границы знакорядов и третей. Z80N заменяет всё это одиночными инструкциями.

Инструкция	Байт	T	Что заменяет
PIXELDN	2	8	Последовательность DOWN_HL из 10+ инструкций (проверка границы трети, обработка переноса, корректировка H и L). Перемещает HL на одну строку пикселей вниз в экранной памяти.
PIXELAD	2	8	Полное вычисление экранного адреса по координатам (D,E). Заменяет подпрограмму pixel_addr (~55T, 15+ инструкций).
SETAE	2	8	Устанавливает соответствующий пиксельный бит в А на основе младших 3 бит Е (x-координата). Заменяет таблицу подстановки или последовательность сдвигов.

С этими тремя инструкциями вся последовательность вывода пикселя, которая на оригинальном Z80 потребляла 70+ тактов и 20+ байтов, становится:

```
pixelad      ; 8T  HL = screen address from (D,E)
setae       ; 8T  A = pixel bit mask from E
or  (hl)    ; 7T  set pixel (non-destructive)
; ... ld (hl), a to write
```

Рендеринг спрайтов (проблема маскированного блита)

Самая CPU-затратная операция в любой игре или демо для Spectrum — маскированный блит спрайта: копирование прямоугольного блока спрайтовых данных в экранную память с пропуском прозрачных пикселей. На стандартном Z80 для этого нужен внутренний цикл из LD/AND/OR/LD на каждый байт, обычно 30–40 тактов на пиксельный байт. Z80N добавляет инструкции блочного копирования со встроенной прозрачностью.

Инструкция	Байт	T	Что заменяет
LDIX	2	16	LDI, но пропускает копирование, если (HL) == A. Однокомандное прозрачное копирование: загрузи в А прозрачный цвет, настрой HL на источник, DE на приёмник — и каждый байт копируется только если он не равен прозрачному значению.
LDDX	2	16	То же, что LDIX, но с декрементом (как LDD).

Инструкция	Байт	Т	Что заменяет
LDIRX	2	21/16	Повторяющийся LDIX. Аппаратный маскированный блит спрайта одной инструкцией. Копирует BC байтов из (HL) в (DE), пропуская любой байт, равный A. 21T на байт при BC>0, 16T на последней итерации.
LDDRX	2	21/16	Повторяющийся LDDX.
LDPIRX	2	21/16	Заливка паттерном с прозрачностью из 8-байтового выровненного источника. Читает по адресу (HL & \$FFF8) + (E & 7), копирует в (DE) если не равно A, инкрементирует DE, декрементирует BC. Аппаратный рендерер тайлового фона. 21T на байт при BC>0, 16T на последней итерации.

Одна LDIRX заменяет самый тщательно оптимизированный внутренний цикл за десятилетия Spectrum-геймдева. LDPIRX ещё экзотичнее — она трактует источник как повторяющийся 8-байтовый паттерн, фактически предостав员я аппаратный тайловый рендерер с прозрачностью. В сочетании с Layer 2 и тайловым оборудованием Next эти инструкции делают ZX Spectrum Next качественно другой платформой для спрайтовых игр.

Арифметика (проблема умножения)

У Z80 нет инструкции умножения. Каждое умножение в коде для Spectrum — это цикл сдвигов и сложений стоимостью 150–250 тактов (см. `mulu112` в Приложении А). Z80N решает это одной инструкцией.

Инструкция	Байт	Т	Что заменяет
MUL D,E	2	8	Беззнаковое умножение 8x8, результат в DE. Заменяет цикл сдвигов и сложений на ~200T.

Восемь тактов. На 28 МГц это 286 наносекунд. Та же операция на стандартном Spectrum с 3,5 МГц занимает примерно 57 микросекунд — улучшение в 200 раз с учётом и более быстрого тактирования, и более быстрой инструкции. Матрицы вращения, преобразования координат, текстурирование, проекция перспективы — всё, что требует умножения, фундаментально дешевле на Z80N.

Битовые манипуляции

Инструкция	Байт	Т	Что заменяет
MIRROR	2	8	Реверсирует все 8 бит А. Горизонтальный переворот спрайта без 256-байтовой таблицы подстановки. На стандартном Z80 реверс битов А требует либо развернутой последовательности из 18 инструкций (LD B,A : XOR A затем 8x RR B : RLA, ~104T), либо 256-байтовой таблицы подстановки (11T, но стоит 256 байтов ОЗУ).
SWAPNIB	2	8	Меняет местами старший и младший nibбл А. Заменяет RLCA : RLCA : RLCA : RLCA (16T, 4 байта).
TEST nn	3	11	AND A, nn без сохранения результата — устанавливает флаги, но сохраняет А. Как СР для побитового AND. Аналог инструкции TST на eZ80.

MIRROR особенно ценна для игр. Без неё каждый горизонтально отражённый спрайт требует либо предварительно отражённой копии в памяти (удвоение спрайтовых данных), либо 256-байтовой таблицы реверса битов плюс побайтовые обращения к таблице. С ней можно отражать спрайты на лету по 8T за байт.

Циклические сдвиги (проблема многобитного сдвига)

На стандартном Z80 сдвиг 16-битного значения более чем на один бит требует цикла: SLA E : RL D на каждый бит, 16T на позицию. Сдвиг DE влево на 5 бит стоит 80T. Z80N добавляет инструкции циклического сдвига, которые сдвигают DE на произвольное число позиций (заданное в В) за постоянное время.

Инструкция	Байт	Т	Что заменяет
BSLA DE,B	2	8	Сдвиг DE влево на В бит. Заменяет B * (SLA E : RL D).
BSRA DE,B	2	8	Арифметический сдвиг DE вправо на В бит (с расширением знака).
BSRL DE,B	2	8	Логический сдвиг DE вправо на В бит (заполнение нулями).
BSRF DE,B	2	8	Сдвиг DE вправо на В бит с заполнением битом 15.
BRLC DE,B	2	8	Циклический сдвиг DE влево на В бит.

Они невероятно полезны в арифметике с фиксированной точкой, субпиксельном позиционировании и любом коде, преобразующем между целочисленными масштабами. Типичная операция вроде «умножить на 5 и сдвинуть вправо на 3», которая на стандартном Z80 заняла бы ~50T, становится тривиальной.

Удобные инструкции

Инструкция	Байт	T	Что заменяет
PUSH nn	4	23	Помещает 16-битное непосредственное значение в стек. Регистр не нужен. Экономит паттерн LD rr, nn : PUSH rr (21T, 4 байта на оригинальном Z80 — тот же размер, на 2T <i>медленнее</i> , но не затрагивает регистровую пару). Когда регистровая пара уже занята, PUSH nn экономит пару PUSH/POP вокруг LD+PUSH, что более чем компенсирует разницу.
ADD HL,A	2	8	Прибавить A к HL. Заменяет последовательность из 5 инструкций на 23T: ADD A,L : LD L,A : ADC A,H : SUB L : LD H,A (или аналогичную через запасной регистр).
ADD DE,A	2	8	То же, что ADD HL,A, но для DE.
ADD BC,A	2	8	То же для BC.
NEXTREG reg, val	4	20	Прямая запись в аппаратный регистр Next. Не нужна настройка портов ввода-вывода. Заменяет LD BC,\$243B : OUT (C),reg : LD BC,\$253B : OUT (C),val — четыре инструкции, 8 байтов, ~48T.
NEXTREG reg,A	3	17	Записать A в аппаратный регистр Next.
OUTINB	2	16	OUT (C),(HL) : INC HL в одной инструкции. Полезна для потоковой передачи данных в порты ввода-вывода.

Общая картина

Z80N — это, по сути, тридцать лет демосценерского разочарования, отлитого в кремнии. Каждая инструкция — это шрам от конкретной, хорошо задокументированной проблемы:

- PIXELDN существует потому, что каждый программист Spectrum хотя бы раз написал DOWN_HL, хотя бы дважды отлаживал случай с границей трети и мечтал больше этого не делать.
- MIRROR существует потому, что каждый игровой программист потратил 256 байтов на таблицу реверса битов для горизонтального отражения спрайтов.
- LDIRX существует потому, что внутренний цикл маскированного блита — это то, на что большинство игр для Spectrum тратят основную часть процессорного времени.
- MUL D,E существует потому, что цикл умножения сдвигом и сложением — самая часто переписываемая подпрограмма в истории Z80.

В отличие от eZ80 (разработанного Zilog для встраиваемых рынков) или R800 (разработанного ASCII Corporation для платформы MSX), Z80N был спроектирован сообществом для сообщества. Набор инструкций читается как список желаний демосцены, потому что он *и есть* список желаний демосцены — команда Next собирала отзывы от действующих кодеров Spectrum и приоритизировала инструкции, которые снимут максимум боли с наиболее частых операций.

3. eZ80 — Корпоративное расширение

eZ80 — это официальный наследник Z80 от Zilog, разработанный для встраиваемых систем, которым нужно больше 64 КБ адресного пространства. Это строгое надмножество Z80 — каждый опкод Z80 допустим и ведёт себя идентично. Расширения скорее архитектурные, чем вычислительные:

- **24-битная адресация и режим ADL.** Регистры могут быть 16-битными или 24-битными в зависимости от рабочего режима. Режим ADL (Address Data Long) даёт 24-битные регистры и плоское адресное пространство на 16 МБ. Z80-совместимый режим ведёт себя точно как стандартный Z80, а регистр MBASE предоставляет недостающие старшие 8 бит адреса.
- **MLT rr — беззнаковое умножение 8x8.** MLT BC умножает B на C и сохраняет 16-битный результат в BC. Аналогично для MLT DE ($D * E \rightarrow DE$) и MLT HL ($H * L \rightarrow HL$). Это гибче, чем MUL D,E на Z80N, который работает только с DE. eZ80 даёт три независимых «умножителя». За 6 тактов на eZ80 (работающем на 18,432 МГц на Agon) это молниеносно быстро.
- **LEA и PEA.** Load Effective Address и Push Effective Address — инструкции вычисления индексированного адреса. LEA rr, IX+d загружает вычисленный адрес в регистровую пару без обращения к памяти. PEA IX+d помещает вычисленный адрес в стек. Полезны для передачи параметров и арифметики указателей.
- **TST (тест) и TSTIO.** Неразрушающий тест AND, аналогичный инструкции TEST nn на Z80N. TSTIO проверяет значение порта ввода-вывода по маске.
- **IN0/OUT0.** Доступ к внутреннему периферийному пространству (адреса 00 — FF).

eZ80 проектировался для промышленного управления, сетевого оборудования и принтеров — не для ретрокомпьютинга. Но он оказался в Agon Light 2, и

внезапно процессор, спроектированный для встраиваемых рынков, стал ретроигровой платформой. Полный справочник по eZ80 — в Приложении Е; история портирования — в Главе 22.

4. R800 — Скоростной монстр MSX turboR

R800 — самый странный член семейства. Разработанный ASCII Corporation для MSX turboR (1990, Panasonic FS-A1GT/FS-A1ST), он Z80-совместим в том смысле, что выполняет полный набор инструкций Z80 — но его внутренняя архитектура радикально отличается.

Конвейер, а не микрокод. Оригинальный Z80 — микрокодовый: каждая инструкция разбивается на машинные циклы (М-цикли) по 3–6 тактов, и сложные инструкции занимают много М-циклов. R800 использует конвейерную архитектуру, где большинство инструкций выполняется за 1–2 тактовых цикла. На 7,16 МГц это даёт эффективную пропускную способность примерно в 5–8 раз выше, чем у Z80 на 3,5 МГц для типичного кода.

Аппаратное умножение. R800 добавляет две инструкции умножения:

Инструкция	Операнды	Результат	Тактов	Примечания
MULUB A, r	A * r (8-бит, без знака)	HL = 16-битное произведение	14	r = B, C, D, E
MULUW HL, rr	HL * rr (16-бит, без знака)	DE:HL = 32-битное произведение	36	rr = BC, SP

Умножение 16x16 с 32-битным результатом за 36 тактов — это впечатляет для процессора эпохи 8 бит. На стандартном Z80 умножение 16x16 занимает 600–1000 тактов в зависимости от реализации. R800 делает 3D-преобразования, DSP-фильтрацию и другие алгоритмы с интенсивным умножением по-настоящему практическими.

Ловушка конвейера. Код, оптимизированный под тайминги Z80, может вести себя неожиданно на R800. Z80-триюк с разверткой LDIR в отдельные LDI (экономящий 5Т на байт) на самом деле работает *медленнее* на R800, потому что конвейер R800 эффективно обрабатывает префикс повтора LDIR. Аналогично, самомодифицирующийся код — фирменный приём демосцены на Z80 — может застопорить конвейер R800, когда запись попадает в предвыбранный инструкцию. Код, быстрый на Z80, не обязательно быстр на R800, и наоборот.

Присутствие на демосцене. У MSX turboR крошечная, но преданная демосцена. Аппаратное умножение делает 3D в реальном времени осуществимым, а чистая скорость позволяет эффекты, невозможные на тактовых частотах Z80. Но редкость платформы (продавалась только в Японии, малая серия) означает, что R800 остаётся сноской в более широкой истории Z80.

5. Советские клоны — За железным занавесом

Советский Союз производил несколько клонов Z80, чтобы обойти западные экспортные ограничения. Эти микросхемы породили целую экосистему ZX Spectrum-совместимых компьютеров, расцветшую с конца 1980-х по 1990-е — и чья демосцена остаётся активной по сей день.

KP1858BM1. Основной советский клон Z80. Совместим по выводам, по инструкциям, по ошибкам. Производился на заводе «Ангстрем» в Зеленограде по реверс-инженерным маскам. KP1858BM1 стоял в основе Pentagon 128 и Scorpion ZS-256 — двух важнейших советских клонов Spectrum и платформ, под которые значительная часть современной российской/СНГ демосцены ZX по-прежнему создаёт свои работы.

T34BM1. Более поздняя CMOS-версия того же дизайна, с меньшим энергопотреблением и слегка отличающимися электрическими характеристиками. Функционально идентичен KP1858BM1.

Ни один из этих чипов не добавляет новых инструкций. Они являются точными функциональными репликами Zilog Z80. Различия электрические: разные технологические процессы, разные временные допуски на установку и удержание, слегка отличающееся поведение на недокументированных опкодах и битах флагов. Для программных целей код, работающий на Zilog Z80, работает идентично на KP1858BM1.

Историческое значение огромно. Без этих клонов экосистема ZX Spectrum не распространилась бы по Советскому Союзу и его государствам-преемникам. Pentagon 128, построенный на KP1858BM1, стал *de-fакто* стандартной платформой Spectrum в России, и его тайминги без спорной памяти (без задержек из-за ULA) — это эталонные тайминги, используемые в этой книге.

6. Сравнительная таблица

Характеристика	Z80	Z80N	eZ80	R800
Тактовая частота (типичная)	3,5 МГц	28 МГц	18,4 МГц	7,16 МГц
Адресное пространство	64 КБ	64 КБ + регистры Next	16 МБ	64 КБ
Аппаратное умножение	Нет	MUL D, E (8T)	MLT rr (6T)	MULUB (14T), MULUW (36T)
Умножение 16x16	Нет	Нет	Нет	MULUW (36T, 32-битный результат)
Циклический сдвиг	Нет	BSLA/BSRA/BSRL/BSRLC DE, B (8T)	BSLA/BSRA/BSRL/BSRLC DE, B (8T)	Нет

Характеристика	Z80	Z80N	eZ80	R800
Блочное копирование с маской	Нет	LDIRX, LDPIRX	Нет	Нет
Помощники экранного адреса	Нет	PIXELDN, PIXELAD, SETAE	Нет	Нет
Реверс битов	Нет	MIRROR (8T)	Нет	Нет
Обмен nibбл	Нет	SWAPNIB (8T)	Нет	Нет
24-битный режим	Нет	Нет	Режим ADL	Нет
Push непосредственного значения	Нет	PUSH nn (23T)	Нет	Нет
Сложение 8-бит с 16-бит	Нет	ADD HL/DE/BC, A (8T)	Нет	Нет
Тест (неразрушающий AND)	Нет	TEST nn (11T)	TST (7T)	Нет
Аппаратный регистровый ввод-вывод	Нет	NEXTREG (17-20T)	IN0/OUT0	Нет
Разработан для	Общего назначения	ZX Spectrum Next	Встраиваемых систем	MSX turboR

Эффективная производительность умножения

Поскольку все четыре варианта работают на разных тактовых частотах, количество тактов само по себе не даёт полной картины. Вот реальное время выполнения беззнакового умножения 8x8 на каждой платформе:

Вариант	Частота	Инструкция	Тактов	Реальное время
Z80	3,5 МГц	Цикл сдвигов и сложений	~200	~57 мкс
Z80N	28 МГц	MUL D,E	8	~0,29 мкс
eZ80	18,4 МГц	MLT DE	6	~0,33 мкс
R800	7,16 МГц	MULUB A,r	14	~1,96 мкс

Z80N и eZ80 по производительности умножения фактически одинаковы. R800 в 30 раз быстрее стандартного Z80, но в 6-7 раз медленнее Z80N/eZ80. Все три достаточно быстры, чтобы сделать 3D в реальном времени практичным.

7. Что это значит для книги

Весь ассемблерный код в этой книге написан для **стандартного набора инструкций Z80**. Каждый пример в каждой главе ассемблируется и запускается на обычном ZX Spectrum 48K, Pentagon 128, Scorpion, MSX или любой другой машине с Zilog Z80 или совместимым клоном. Никаких расширений не требуется.

Это осознанный выбор. Принципы оптимизации — бюджет тактов, распределение регистров, структура циклов, самомодифицирующийся код, развернутые циклы, трюки со стеком — универсальны. Код, быстрый на Z80 с 3,5 МГц, быстр и на Z80N с 28 МГц. Код, помещающийся в 48 КБ, помещается и в 16 МБ. Ограничения оригинального Z80 учат тебя мыслить так, что это переносится на каждый вариант семейства.

Тем не менее, если у тебя есть ZX Spectrum Next, расширения Z80N слишком хороши, чтобы их игнорировать. Глава 22 охватывает стратегии портирования, включая Z80N-специфичные оптимизации. Если у тебя Agon Light 2, Приложение E — твой справочник по eZ80, а Глава 22 проведёт через полныйпорт со Spectrum на Agon. Циклические сдвиги, аппаратное умножение и инструкции маскированного бита не меняют как ты думаешь об оптимизации — они меняют *где переместится узкое место*, когда классические болевые точки будут устранены.

Фундаментальные принципы не меняются. Инструкции становятся лучше.

См. также

- **Приложение А: Краткий справочник инструкций Z80** — полная таблица стандартных инструкций Z80 с тактами, размерами в байтах и влиянием на флаги. Базовая линия, общая для всех вариантов.
- **Приложение Е: Краткий справочник по eZ80** — полный справочник по eZ80, включая систему режимов, MLT, LEA/PEA и особенности Agon Light 2.
- **Глава 22: Портрорование — Agon Light 2** — практическое пошаговое руководство по портированию, охватывающее расширения eZ80 и Z80N в контексте.

Примечание по аудиту тактов: Значения тактов Z80N в этом приложении были исправлены по официальной документации на wiki.specnext.dev. Изначальные значения (приблизительно заниженные вдвое по всем позициям) были выявлены как некорректные Ped7g (Петер Хелцмановски).

Источники: 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)

Приложение G: Справочник регистров AY-3-8910 / TurboSound / Triple AY

«Четырнадцать регистров. Три канала прямоугольного тона. Один генератор шума. Один генератор огибающей. Это всё, что у тебя есть.» – Глава 11

Это приложение – полный справочник на уровне регистров для программируемого звукового генератора AY-3-8910, используемого в ZX Spectrum 128K, клонах с TurboSound (Pentagon, Scorpion) и конфигурации тройного AY на ZX Spectrum Next. Глава 11 покрывает музыкальные техники; это приложение – даташит, который ты держишь открытым при написании кода.

Все шестнадцатеричные значения используют нотацию \$FF. Двоичные – %10101010. Частота тактирования предполагается стандартной PAL Spectrum 128K – 1,7734 МГц, если не указано иное.

Порты ввода/вывода на ZX Spectrum 128K

Порт	Направление	Функция
FFFFD * (0 – 15) * *FFFD	Чтение	Чтение значения текущего выбранного регистра
\$BFFD	Запись	Запись данных в выбранный регистр

Последовательность записи

Каждая запись в регистр AY – двухшаговая операция: выбрать регистр, затем записать значение.

```
; 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

```

Трюк: между двумя инструкциями OUT меняется только старший байт BC. Младший байт \$FD остаётся в C. Это экономит 3 байта и 10 тактов (T-state) по сравнению с загрузкой полного 16-битного значения дважды.

Чтение регистра

```

; Clobbers: BC
ay_read:
    ld bc, $FFFFD
    out (c), a      ; select register
    in a, (c)        ; read value
    ret

```

Чтение полезно для сохранения состояния микшера (R7) при изменении флагов включения отдельных каналов.

Массовая запись регистров

Для музыкальных движков, обновляющих все 14 регистров за кадр, развернутый цикл – самый быстрый:

```

; 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 = $FFFFD (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 = $FFFFD
    inc a
    cp 14
    jr nz, .loop
    ret

```

Полная карта регистров

Обзор

Рег	Имя	Используемые биты	Чт/Зп	Сброс	Описание
R0	Период тона А, мл.	7-0	Чт/Зп	\$00	Период тона канала А, биты 7-0

594ПРИЛОЖЕНИЕ G: СПРАВОЧНИК РЕГИСТРОВ AY-3-8910 / TURBOSOUND / TRIPLE AY

Reg	Имя	Используемые биты	Чт/Зп	Сброс	Описание
R1	Период тона А, ст.	3-0	Чт/Зп	\$00	Период тона канала А, биты 11-8
R2	Период тона В, мл.	7-0	Чт/Зп	\$00	Период тона канала В, биты 7-0
R3	Период тона В, ст.	3-0	Чт/Зп	\$00	Период тона канала В, биты 11-8
R4	Период тона С, мл.	7-0	Чт/Зп	\$00	Период тона канала С, биты 7-0
R5	Период тона С, ст.	3-0	Чт/Зп	\$00	Период тона канала С, биты 11-8
R6	Период шума	4-0	Чт/Зп	\$00	Период генератора шума (0-31)
R7	Микшер / I/O	7-0	Чт/Зп	\$FF	Включение тона/шума + направление портов I/O
R8	Громкость А	4-0	Чт/Зп	\$00	Громкость канала А или режим огибающей
R9	Громкость В	4-0	Чт/Зп	\$00	Громкость канала В или режим огибающей
R10	Громкость С	4-0	Чт/Зп	\$00	Громкость канала С или режим огибающей
R11	Период огибающей, мл.	7-0	Чт/Зп	\$00	Период огибающей, биты 7-0
R12	Период огибающей, ст.	7-0	Чт/Зп	\$00	Период огибающей, биты 15-8
R13	Форма огибающей	3-0	Зп	-	Форма волны огибающей (запись перезапускает огибающую)
R14	Порт I/O А	7-0	Чт/Зп	-	Универсальный I/O (прямое подключение на AY-3-8910)

Рег	Имя	Используемые биты	Чт/Зп	Сброс	Описание
R15	Порт I/O B	7-0	Чт/Зп	-	Универсальный I/O (прямое подключение на AY-3-8910)

Замечание по R14-R15: AY-3-8910 имеет два 8-битных порта I/O. На ZX Spectrum 128K порт I/O A (R14) активен – он читает матрицу клавиатуры и другие аппаратные сигналы. R15 (порт I/O B) обычно не подключён. AY-3-8912 (используемый в некоторых клонах) имеет только порт A; AY-3-8913 вообще не имеет портов I/O. Для программирования звука R14 и R15 не имеют значения.

R0-R1: Период тона канала А (12 бит)

```
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
```

- **Диапазон:** 1 до 4095 (12-битное беззнаковое). Период 0 ведёт себя как 1 на большинстве реализаций.
- **Действие:** Устанавливает частоту прямоугольной волны на канале А.
- **Формула:** $\text{frequency} = 1773400 / (16 * \text{period})$ (PAL Spectrum 128K)
- **Практический диапазон:** Период 1 = 110 837 Гц (ультразвук), период 4095 = 27 Гц (глубокий бас).

```
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: Период тона канала В (12 бит)

Раскладка идентична R0-R1, для канала В.

R4-R5: Период тона канала С (12 бит)

Раскладка идентична R0-R1, для канала С.

R6: Период шума (5 бит)

```
R6: [ 0] [ 0] [ 0] [D4] [D3] [D2] [D1] [D0]
```

- **Диапазон:** 0 до 31. Единственный генератор шума, общий для всех трёх каналов.
- **Действие:** Меньшие значения = более высокочастотный шум (шипение). Большие значения = более низкочастотный, грубый шум.

- **Формула:** noise_frequency = 1773400 / (16 * period) (та же, что для тона)

Значение R6	Характер	Типичное применение
0-5	Высокое шипение	Хай-хэт, тарелка, металлический призвук
6-12	Средний шум	Тело малого барабана, белый шум
13-20	Низкий гул	Взрыв, двигатель, ветер
21-31	Очень низкий	Гром, далёкий гул

Генератор шума использует 17-битный регистр сдвига с линейной обратной связью (LFSR) для генерации псевдослучайного выхода. Выход одинаков для AY-3-8910 и YM2149, но позиции отводов LFSR различаются, создавая чуть разные текстуры шума на каждом чипе.

R7: Управление микшером (самый важный регистр)

R7: [IOB] [IOA] [NC] [NB] [NA] [TC] [TB] [TA]
bit7 bit6 bit5 bit4 bit3 bit2 bit1 bit0

Бит	Имя	0 =	1 =
7	Направление порта I/O B	Выход	Вход
6	Направление порта I/O A	Выход	Вход
5	Включение шума C	ВКЛ	Выкл
4	Включение шума B	ВКЛ	Выкл
3	Включение шума A	ВКЛ	Выкл
2	Включение тона C	ВКЛ	Выкл
1	Включение тона B	ВКЛ	Выкл
0	Включение тона A	ВКЛ	Выкл

Критически важно: **0** означает **ВКЛ**. Микшер использует активно-низкую логику. Сброшенный бит включает соответствующий источник. Это самый запутывающий аспект программирования AY.

Биты 7-6: Всегда устанавливай в 1 (режим входа) на Spectrum. Не меняй их, если ты не знаешь, что делаешь с портами I/O.

Типичные значения микшера

Значение	Двоичное (NC NB NA TC TB TA)	Эффект
\$38	%00 111 000	Все три тона, без шума
\$3E	%00 111 110	Только тон A
\$3D	%00 111 101	Только тон B
\$3B	%00 111 011	Только тон C
\$36	%00 110 110	Тон A + Шум A
\$2D	%00 101 101	Тон B + Шум B
\$1B	%00 011 011	Тон C + Шум C
\$28	%00 101 000	Все тона + Шум C (музыка + барабаны на C)

Значение	Двоичное (NC NB NA TC TB TA)	Эффект
\$07	%00 000 111	Весь шум, без тонов
\$00	%00 000 000	Всё включено
\$3F	%00 111 111	Всё выключено (тишина)

Примечание: Двоичные значения выше показывают только биты 5-0. Биты 7-6 должны быть %11 для нормальной работы (порты I/O как входы), что делает, например, \$38 реально %11 111 000 = \$F8. Однако на Spectrum 128K неиспользуемые старшие биты игнорируются по соглашению, и большинство кода использует короткую форму. Некоторые движки записывают полную форму \$F8; оба варианта работают идентично.

```
; 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: Регистры громкости (5 бит)

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

Бит	Функция
4 (M)	Режим: 0 = фиксированная громкость (биты 3-0), 1 = использовать генератор огибающей
3-0	Уровень фиксированной громкости: 0 (тишина) до 15 (максимум)

- Режим фиксированной громкости** (бит 4 = 0): Биты 3-0 задают громкость канала напрямую. 15 = самый громкий, 0 = тишина. Кривая громкости логарифмическая на подлинном AY-3-8910 (приблизительно 1,5 дБ на шаг), но линейная на YM2149.
- Режим огибающей** (бит 4 = 1): Громкость канала управляет генератором огибающей (R11-R13). Биты 3-0 игнорируются. Существует только один генератор огибающей, общий для всех каналов в режиме огибающей.

Значение	Эффект
\$00	Тишина
\$08	Примерно половина громкости
\$0F	Максимальная фиксированная громкость
\$10	Режим огибающей (любое значение \$10-\$1F активирует огибающую)

Трюк с 4-битным ЦАП: Быстро записывая последовательные значения сэмплов в регистр громкости (без режима огибающей), можно воспроизводить

оцифрованный звук через AY. При частоте дискретизации 8 кГц это потребляет примерно 437 тактов (T-state) на запись сэмпла, не оставляя процессору почти ничего для другой работы. Подробнее о технике гибридных барабанов в главе 12.

```

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: Период огибающей (16 бит)

R11: [D7] [D6] [D5] [D4] [D3] [D2] [D1] [D0] bits 7-0
R12: [D7] [D6] [D5] [D4] [D3] [D2] [D1] [D0] bits 15-8

- **Диапазон:** 1 до 65535 (16-битное беззнаковое). Период 0 ведёт себя как 1.
- **Формула:** envelope_frequency = 1773400 / (256 * period)
- **При периоде = 1:** ~6 927 Гц (один полный цикл огибающей за ~144 микросекунды)
- **При периоде = 65535:** ~0,106 Гц (один цикл за ~9,4 секунды)

Период огибающей управляет скоростью нарастания или спада громкости в соответствии с формой, выбранной в R13. Для buzz-bass период огибающей заменяет период тона как управление высотой:

bass_envelope_period = 1773400 / (256 * desired_frequency)

Для баса C2 (65,4 Гц): период = 1773400 / (256 * 65,4) = 106.

```

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: Форма огибающей (4 бита, только запись)

R13: [0] [0] [0] [0] [CONT] [ATT] [ALT] [HOLD]
bit3 bit2 bit1 bit0

Бит	Имя	Функция
3	CONT	Продолжение: 0 = однократная (сброс в 0 или удержание), 1 = повторяющаяся

Бит	Имя	Функция
2	ATT	Атака: 0 = начать с макс., спад вниз, 1 = начать с 0, нарастание вверх
1	ALT	Чередование: 0 = одно направление каждый цикл, 1 = реверс направления
0	HOLD	Удержание: 0 = нормальный цикл, 1 = удержание на финальном уровне после первого цикла

Критическое поведение: Запись ЛЮБОГО значения в R13 немедленно перезапускает огибающую с начала цикла. Это верно, даже если ты записываешь то же значение, что уже хранится. Это поведение перезапуска критически важно для запуска нот buzz-bass и звуков ударных.

Все 16 значений формы

Хотя 4 бита позволяют 16 значений, многие дают идентичный результат. Уникальных форм всего 10:

Значение	Биты (CONT ATT ALT HOLD)	Форма	Поведение
\$00	0 0 0 0	__	Спад до 0, удержание на 0
\$01	0 0 0 1	__	То же, что \$00
\$02	0 0 1 0	__	То же, что \$00
\$03	0 0 1 1	__	То же, что \$00
\$04	0 1 0 0	/__	Атака до 15, сброс до 0, удержание на 0
\$05	0 1 0 1	/__	То же, что \$04
\$06	0 1 1 0	/__	То же, что \$04
\$07	0 1 1 1	/__	То же, что \$04
\$08	1 0 0 0	\///	Повторяющаяся пила вниз
\$09	1 0 0 1	__	Однократный спад, удержание на 0 (то же, что \$00)
\$0A	1 0 1 0	\//\	Повторяющийся треугольник (вниз-вверх)
\$0B	1 0 1 1	\^^\	Однократный спад, затем удержание на макс. (15)

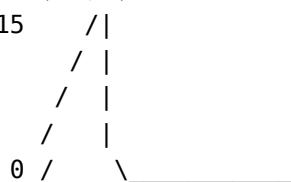
Значение	Биты (CONT ATT ALT HOLD)	Форма	Поведение
\$0C	1 1 0 0	////	Повторяющаяся пила вверх
\$0D	1 1 0 1	/^^	Однократная атака, затем удержание на макс. (15)
\$0E	1 1 1 0	/\/\	Повторяющийся треугольник (вверх-вниз)
\$0F	1 1 1 1	/__	Однократная атака, сброс до 0, удержание на 0 (то же, что \$04)

Формы волн огибающей

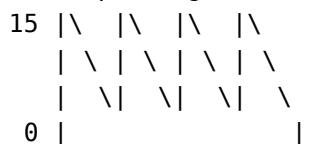
\$00-\$03, \$09:



\$04-\$07, \$0F:



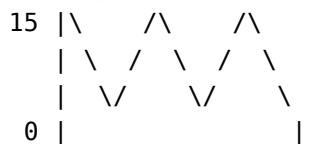
\$08 (repeating \\\\"":



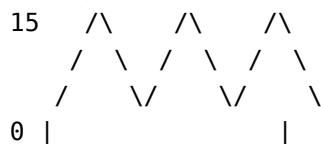
\$0C (repeating /////":



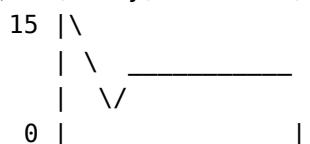
\$0A (repeating /\//):



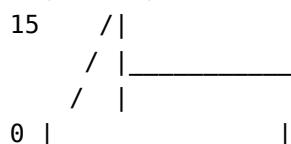
\$0E (repeating /\/\):



\$0B (decay, hold max):



\$0D (attack, hold max):



Практические применения огибающей

Форма	Значение	Сценарий использования
\$00	__	Затухание барабана: резкий удар, затухающий до тишины
\$08	\ \	Buzz-bass: густой повторяющийся басовый тон
\$0C	///	Buzz-bass (инвертированная фаза): та же высота, другой тембр
\$0A	\//\//	Металлический/колокольный тон (треугольная модуляция)
\$0E	/\//\	Металлический/колокольный тон (инвертированный треугольник)
\$0D	/^\^\^	Нарастание: громкость поднимается до макс. и удерживается

Преобразование периода тона в частоту

Формула

```
frequency = AY_clock / (16 * period)
period      = AY_clock / (16 * frequency)
```

Тактовая частота AY по платформам

Платформа	Частота AY	Частота CPU	Соотношение
ZX Spectrum 128K (PAL)	1 773 400 Гц	3 546 900 Гц	AY = CPU / 2
ZX Spectrum 128K (NTSC)	1 789 772 Гц	3 579 545 Гц	AY = CPU / 2
Pentagon 128	1 750 000 Гц	3 500 000 Гц	AY = CPU / 2
Amstrad CPC	1 000 000 Гц	4 000 000 Гц	AY = CPU / 4
MSX	1 789 772 Гц	3 579 545 Гц	AY = CPU / 2
Atari ST (YM2149)	2 000 000 Гц	8 000 000 Гц	YM = CPU / 4
ZX Spectrum Next	1 773 400 Гц	варьируется	То же, что 128K

Практическое следствие: Одно и то же значение периода даёт чуть разную высоту на разных платформах. Мелодия, сочинённая на Pentagon (1,75 МГц), будет звучать чуть выше на Spectrum 128K (1,7734 МГц). Для большинства музыкальных целей разница неслышима.

Полная таблица частот нот

Все значения периодов рассчитаны для тактовой частоты AY = 1 773 400 Гц (PAL Spectrum 128K).

Октыавы 1-2 покрывают басовый диапазон (территория buzz-bass). Октыавы 3-6 – основной мелодический диапазон. Октыавы 7-8 – высокочастотные, с нарастающей неточностью из-за целочисленного округления.

Октыава 1 (глубокий бас)

Нота	Частота (Гц)	Период	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

Октыава 2 (бас)

Нота	Частота (Гц)	Период	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

Октыава 3

Нота	Частота (Гц)	Период	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

Нота	Частота (Гц)	Период	R_LO	R_HI
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

Октава 4 (средняя октава)

Нота	Частота (Гц)	Период	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

Октава 5

Нота	Частота (Гц)	Период	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

Октава 6

Нота	Частота (Гц)	Период	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

Нота	Частота (Гц)	Период	R_LO	R_HI
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

Октава 7 (высокая)

Нота	Частота (Гц)	Период	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

Окта́ва 8 (очень высокая - ограниченная точность)

Нота	Частота (Гц)	Период	R_LO	R_HI	Реальная частота	Ошибка (центы)
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

Примечание: Выше октавы 6 целочисленное округление значения периода даёт всё более слышимые ошибки высоты. Столбцы «Реальная частота» и «Ошибка» для октавы 8 показывают реальную выходную частоту и отклонение в центах. Для высоких нот тонкая настройка невозможна – разрешение просто слишком грубое.

Компактная таблица нот для Z80-кода

Большинство движков трекеров хранят таблицу периодов как 16-битные слова. Вот практическая таблица на 96 нот (октавы 1-8) для включения в ассемблер-

ный исходник:

```
; 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

; 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
```

Таблица #5: Натуральный строй (чистая интонация)

Стандартная таблица нот выше использует равномерную темперацию (12-ТET) – каждый полутон – это корень 12-й степени из 2. Это хорошо работает для тональных каналов, но создаёт проблему для **buzz-bass (T+E)**: поскольку $\text{envelope_period} = \text{tone_period} / 16$, любой период тона, не делящийся на 16, вносит ошибку округления. Огибающая дрейфует относительно тона, создавая слышимые биения на длинных басовых нотах.

«Частотная таблица с нулевой ошибкой» Ивана Рошина (2001) и реализация

oisee для VTi (2009) решают это, используя **натуральный строй** – целочисленные интервалы для до-мажора / ля-минора:

C [9/8] D [10/9] E [16/15] F [9/8] G [10/9] A [9/8] B [16/15] C

Это даёт чистые квинты (C-G, E-B, A-E в точном соотношении 3:2) и, что критически важно, периоды, где большинство нот основной гаммы делятся на 16 нацело.

Тактовая частота AY: 1 520 640 Гц (нестандартная; выбирай частоту для нужной тональности ниже):

```
; 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, 13, 12
```

Проверка делимости периодов (октава 2, басовый диапазон):

Нота	Период	mod 16	Период огибающей	Чистый Т+Е?
C2	1440	0	90	Да
C#2	1350	6	84	Нет
D2	1280	0	80	Да
D#2	1200	0	75	Да
E2	1152	0	72	Да
F2	1080	0	67	~
F#2	1013	5	63	Нет
G2	960	0	60	Да
G#2	900	4	56	Нет
A2	864	0	54	Да
A#2	810	2	50	Нет

Нота	Период	mod 16	Период огибающей	Чистый Т+Е?
B2	768	0	48	Да

Семь из двенадцати нот (все натуральные ноты до-мажора) делятся нацело – по сравнению с *нулём* в равнотемперированной таблице.

Транспозиция через частоту чипа: поскольку таблица фиксирована для до/ля минор, другие тональности требуют другой тактовой частоты AY. Каждый шаг умножает на $2^{(1/12)}$:

Тональность	Частота чипа (Гц)
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

На реальном железе тактовая частота AY фиксирована; в трекерах (Vortex Tracker II/Improved) и эмуляторах это настройка модуля. Таблица #5 – стандарт для конвертера autosiril MIDI-to-PT3, потому что большинство конвертированных треков активно используют buzz-bass. Подробное объяснение проблемы выравнивания Т+Е в главе 11.

TurboSound: 2 x AY

TurboSound – аппаратная модификация, добавляющая второй чип AY, что даёт 6 тональных каналов, 2 генератора шума и 2 генератора огибающей. Наиболее распространённая современная реализация – карта NedoPC TurboSound.

Выбор чипа

Оба чипа AY делят одни и те же порты I/O (FFF/FFD). Значение выбора чипа, записанное в \$FFF, переключает все последующие операции с регистрами на выбранный чип.

Значение выбора чипа	Цель
\$FF	Чип 0 (основной / оригинальный AY)
\$FE	Чип 1 (дополнительный AY)

```

ld bc, $FFFF
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

```

Важно: Выбор чипа сохраняется до изменения. После выбора чипа 1 все операции с регистрами – включая чтение – направляются на чип 1. Всегда явно выбирай чип перед любым обращением к регистрам в своём движке.

Архитектура движка TurboSound

Типичный музыкальный движок TurboSound поддерживает два 14-байтных буфера регистров и записывает их последовательно:

```

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

```

Общая стоимость: примерно 28 инструкций OUT, порядка 700-800 тактов (T-state) включая накладные расходы. Это около 1% бюджета кадра – пренебрежимо мало.

Стерео-конфигурация

TurboSound позволяет настоящее стерео. Два чипа можно жёстко панорамировать или смешивать:

Конфигурация	Чип 0 (левый)	Чип 1 (правый)	Характер
Широкое стерео	Лид, бас (A0+B0), барабаны (C0)	Контрмелодия (A1+B1), пэды (C1)	Просторное, концертное
Центрированный бас	Бас на C0 + C1 (те же данные)	Лид на A0, гармония на A1	Плотный низ

Конфигурация	Чип 0 (левый)	Чип 1 (правый)	Характер
Раздельные барабаны	Бочка (C0)	Малый + хай-хэт (C1)	Мощная, широкая перкуссия

У каждого чипа свой независимый генератор шума и генератор огибающей. Это значит, что можно запустить buzz-bass на чипе 0 и совершенно независимую огибающую для перкуссии на чипе 1 без интерференции – невозможно на одном AY.

ZX Spectrum Next: Triple AY (3 x AY)

ZX Spectrum Next включает три AY-совместимых звуковых чипа, обеспечивая 9 тональных каналов, 3 генератора шума и 3 генератора огибающей.

Выбор чипа на Next

Next расширяет протокол TurboSound. Третий чип выбирается тем же механизмом:

Значение выбора чипа	Цель
\$FF	Чип 0
\$FE	Чип 1
\$FD	Чип 2

Next также предоставляет доступ через Next-регистр \$06 (Peripheral 2), настраивающий режим звукового чипа:

Next Reg \$06, биты 1-0	Режим
%00	Одиночный AY (совместимость со Spectrum 128K)
%01	TurboSound (2 x AY)
%10	Triple AY (3 x AY)

Стерео-панорамирование каждого канала

Next добавляет функцию, которой у оригинального AY никогда не было: стереопанорамирование для каждого канала. Каждый из 9 каналов может быть индивидуально панорамирован влево, вправо или в центр.

Панорамирование управляет через регистр AY 14 (R14), переназначенный на Next как регистр стерео-управления при обращении в контексте чипа AY:

Бит	Канал	0 =	1 =
0	A, правый	выкл	вкл
1	A, левый	выкл	вкл

Бит	Канал	0 =	1 =
2	B, правый	выкл	вкл
3	B, левый	выкл	вкл
4	C, правый	выкл	вкл
5	C, левый	выкл	вкл

Установи оба бита (левый + правый) для центрального панорамирования. Установи только один бит для жёсткого левого или правого.

Типичные паттерны в Z80-коде

Заглушить AY

```
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
```

Воспроизвести одну ноту на канале А

```
; 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
```

Запустить ноту buzz-bass

```
; 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
```

Перезапуск огибающей

```
; 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
```

Удар цифрового барабана (4-битный ЦАП)

```
; 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
```

Подробнее о технике гибридных барабанов (цифровая атака + затухание огибающей AY), делающей это практическим в движке демо, в главе 12.

Замечания по форматам трекеров

ProTracker 3 (.pt3)

ProTracker 3 (PT3) – наиболее широко используемый формат трекеров на ZX Spectrum 128K. Кросс-платформенный редактор **Vortex Tracker II** нативно создаёт файлы .pt3.

Свойство	Значение
Каналы	3 (одиночный AY) или 6 (TurboSound)
Длина паттерна	1-64 строки
Скорость песни	1-255 (кадров на строку)
Инструменты	Орнаментные таблицы + амплитудные огибающие
Эффекты	Арпеджио, портаменто, вибрето, управление огибающей
Размер проигрывателя	~1,2-1,8 КБ (ассемблер Z80)
Формат данных	Упакованные паттерны + таблицы инструментов/орнаментов

Как данные PT3 отображаются на регистры AY

На каждом кадре (1/50 секунды) проигрыватель PT3:

1. Декрементирует счётчик скорости. Если он достигает нуля, переходит к следующей строке.
2. Для каждого канала (A, B, C):
 - Читает ноту, инструмент и эффект из текущей строки.
 - Применяет орнамент инструмента (покадровая таблица смещений высоты тона).
 - Применяет амплитудную огибающую инструмента (покадровая таблица громкости).
 - Вычисляет итоговый период тона и громкость.
3. Обновляет регистр микшера (R7) на основе того, какие каналы используют тон, шум или оба.
4. Записывает все вычисленные значения в R0-R13.

PT3 row data:

```
Note C-4, Instrument 3, Effect: Arpeggio +4+7
|           |           |
|           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 -> ...)
```

Интеграция в твой проект

```
; 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
```

Другие форматы трекеров

Формат	Трекер	Каналы	Размер проигрывателя	Ключевая особенность
.pt3	Pro Tracker 3 / Vortex Tracker II	3-6	~1,2-1,8 КБ	Стандарт индустрии. Орнаменты + сэмплы.
.pt2	Pro Tracker 2	3	~0,8-1,0 КБ	Старше, проще. Всё ещё используется из-за размера.
.stc	Sound Tracker / Sound Tracker Pro	3	~0,5-0,7 КБ	Старейший формат Spectrum. Без орнаментов.

Формат	Трекер	Каналы	Размер проигрывателя	Ключевая особенность
.asc	ASC Sound Master	3	~0,8-1,0 КБ	Компактный проигрыватель. Фаворит российской сцены.
.sqt	SQ-Tracker	3	~0,6-0,8 КБ	Отличное сжатие данных.
.ay	AY emulation container	разное	N/A	Захватывает сырье дампы регистров. Для эмуляторов, не для движков воспроизведения.

Vortex Tracker II

Vortex Tracker II – современный стандарт для сочинения музыки AY. Работает на Windows (и через Wine на Linux/macOS) и напрямую экспортирует файлы .pt3, совместимые со всеми стандартными Z80-проигрывателями.

Ключевые возможности для демосцены: - **Режим TurboSound:** Редактирование 6 каналов (2 x AY) в одном модуле. - **Редактор орнаментов:** Визуальные покадровые таблицы смещений высоты тона. - **Редактор сэмплов:** Покадровое управление амплитудой + тоном/шумом. - **Точки петли:** Установка начала петли для каждого паттерна и для всей песни. - **Экспорт:** .pt3 (нативный), .txt (текстовый дамп для анализа), .wav (аудио-рендер).

Типичный рабочий процесс демосцены: 1. Сочиняй в Vortex Tracker II. 2. Экспортируй как .pt3. 3. Подключи стандартный pt3player.asm (широко доступен, множество версий, оптимизированных по размеру или скорости). 4. INCBIN данные .pt3. 5. Вызывай music_init и music_play, как показано выше.

AY-3-8910 vs YM2149: различия, которые имеют значение

Yamaha YM2149 (используемый в Atari ST и некоторых клонах Spectrum) совместим по выводам с AY-3-8910, но не побитно идентичен:

Особенность	AY-3-8910	YM2149
Кривая громкости	Логарифмическая (1,5 dB/шаг)	Линейная
LFSR шума	17-бит, специфические отводы	17-бит, другие отводы
Точность огибающей	16 уровней громкости (4-бит)	32 уровня громкости (5-бит внутренне)

Особенность	AY-3-8910	YM2149
Вывод 26 (SEL)	Делитель тактовой частоты	То же, но часто зашит аппаратно
Выходной ЦАП	4-битная резистивная лестница	5-битная резистивная лестница

Практическое влияние: Одна и та же мелодия звучит теплее/басовитее на настоящем AY-3-8910 и ярче/тоньше на YM2149 из-за различных кривых громкости. Эмуляторы обычно позволяют выбрать, какой чип эмулировать. При тестировании своей музыки попробуй оба – у твоей аудитории может быть любой чип в машине.

Краткая справочная карточка

Сводка регистров (отрывная)

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: \$FFFFD = select register \$BFFD = write data

Write: OUT (\$FFFFD),reg then OUT (\$BFFD),value

Read: OUT (\$FFFFD),reg then IN A,(\$FFFFD)

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->\$FFFFD = chip 0 \$FE->\$FFFFD = chip 1

Next 3xAY: \$FF = chip 0 \$FE = chip 1 \$FD = chip 2

Источники: 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); NedоПC TurboSound FM documentation; ZX Spectrum Next User Manual, Issue 2

Приложение Н: API хранилищ — TR-DOS и esxDOS

«Технически впечатляющее демо, которое поставляется как .tzx, когда правила требуют .trd, будет дисквалифицировано.» – Глава 20

Два API хранилищ доминируют в мире ZX Spectrum: **TR-DOS** (дисковая операционная система советского интерфейса Beta Disk 128, стандартная на клонах Pentagon и Scorpion) и **esxDOS** (современная операционная система SD-карт, работающая на аппаратных интерфейсах DivMMC и DivIDE). Большинство релизов российской и украинской демосцены распространяются как образы дисков .trd. Большинство современных западных релизов используют образы лент .tap или файловые структуры, совместимые с esxDOS. Если ты выпускаешь демо или игру сегодня, практичный выбор — предоставить образ .trd для совместимости с огромной российско-украинской базой пользователей, и файл .tap для всех остальных. Если твой загрузчик поддерживает определение esxDOS (как описано в Главе 21), пользователи с DivMMC получают быструю загрузку с SD-карты бесплатно.

Это приложение — справочник по API, который ты держишь открытым, пока пишешь загрузчик. Глава 21 описывает интеграцию в полноценном игровом проекте. Глава 15 описывает аппаратные детали переключения банков памяти и маппинга портов, на которых основаны оба API.

1. TR-DOS (Beta Disk 128)

Аппаратная часть

Интерфейс Beta Disk 128 — стандартный контроллер гибких дисков для Pentagon, Scorpion и большинства советских клонов ZX Spectrum. Он построен на микросхеме контроллера гибких дисков Western Digital WD1793, которая взаимодействует с Z80 через пять портов ввода-вывода.

ПЗУ TR-DOS (8 КБ) занимает диапазон \$0000-\$3FFF, когда интерфейс Beta Disk активен. Оно подключается автоматически, когда Z80 выполняет код по адресу \$3D13 (магическая точка входа), и отключается, когда выполнение возвращается в область основного ПЗУ.

Формат диска

Свойство	Значение
Дорожки	80
Стороны	2
Секторов на дорожку	16
Байт на сектор	256
Общая ёмкость	640 КБ (655 360 байт)
Формат образа	.trd (сырой образ диска, 640 КБ)
Системная дорожка	Дорожка 0, сторона 0

Дорожка 0 содержит каталог диска (секторы 1-8) и информационный сектор диска (сектор 9). Каталог вмещает до 128 записей о файлах. Каждая запись занимает 16 байт:

```
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
```

Карта портов WD1793

Порт	Чтение	Запись
\$1F	Регистр состояния	Регистр команд
\$3F	Регистр дорожки	Регистр дорожки
\$5F	Регистр сектора	Регистр сектора
\$7F	Регистр данных	Регистр данных
\$FF	Системный регистр TR-DOS	Системный регистр TR-DOS

Порт \$FF — системный порт Beta Disk. Он управляет выбором дисковода, выбором стороны, загрузкой головки и плотностью записи. Старшие биты также отражают сигналы DRQ (Data Request) и INTRQ (Interrupt Request) от WD1793.

Команды WD1793

Команда	Код	Описание
Restore	\$08	Переместить головку на дорожку 0. Проверить дорожку.
Seek	\$18	Переместить головку на дорожку из регистра данных.
Step In	\$48	Шаг головки на одну дорожку к центру.
Step Out	\$68	Шаг головки на одну дорожку к краю.

Команда	Код	Описание
Read Sector	\$88	Прочитать один 256-байтный сектор.
Write Sector	\$A8	Записать один 256-байтный сектор.
Read Address	\$C0	Прочитать следующее поле ID сектора.
Force Interrupt	\$D0	Прервать текущую команду.

Младший nibбл каждого байта команды содержит флаги-модификаторы (скорость шага, проверка, выбор стороны, задержка). Значения выше используют общие значения по умолчанию. За полной битовой раскладкой обращайся к даташиту WD1793.

API ПЗУ: загрузка файла

Стандартный подход к файловому вводу-выводу под TR-DOS — вызов подпрограмм ПЗУ по адресу \$3D13. ПЗУ TR-DOS предоставляет высокоуровневые файловые операции через систему команд: ты помещаешь параметры в регистры и в системную область по адресам \$5D00–\$5FFF, затем вызываешь ПЗУ.

```
; 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
```

Чтобы загрузить по конкретному адресу (переопределив сохранённый начальный адрес):

```
; 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

```

Прямой доступ к секторам

Для демо, которые потоково читают данные с диска — полноэкранные анимации, музыкальные данные, не помещающиеся в доступную оперативную память, или многочастные демо, подгружающие эффекты на лету — прямой доступ к секторам полностью обходит файловую систему. Ты управляешь положением головки, читаешь секторы по одному и обрабатываешь данные по мере их поступления.

```

; 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

```

Внимание: Прямой доступ к секторам чувствителен к таймингам. Прерывания должны быть запрещены во время цикла передачи данных, иначе байты будут потеряны. WD1793 выставляет DRQ на ограниченное временное окно; если Z80 не прочитает регистр данных до прихода следующего байта, данные будут

перезаписаны. На скорости 250 кбит/с (двойная плотность) у тебя примерно 32 микросекунды на байт — около 112 тактов (T-state) на Pentagon. Плотный цикл выше выполняется примерно за 50–60 тактов (T-state) на байт, оставляя достаточный запас.

Обнаружение диска

Чтобы определить, присутствует ли интерфейс Beta Disk:

```
; 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
```

Более надёжный метод — попытаться вызвать \$3D13 и проверить, присутствуют ли сигнатурные байты ПЗУ TR-DOS. В продакшн-коде обычно проверяют известную последовательность байт в точках входа ПЗУ TR-DOS.

2. esxDOS (DivMMC / DivIDE)

Аппаратная часть

DivMMC (и его более старый собрат DivIDE) — интерфейс массового хранилища, подключающий SD-карту к ZX Spectrum. Прошивка esxDOS предоставляет POSIX-подобный файловый API, доступный из кода Z80 через RST \$08. esxDOS поддерживает файловые системы FAT16 и FAT32, длинные имена файлов, подкаталоги и множество одновременно открытых файловых дескрипторов.

DivMMC использует автоматическое маппирование: когда Z80 считывает инструкцию с определённых «ловушечных» адресов (в частности \$0000, \$0008, \$0038, \$0066, \$04C6, \$0562), аппаратура DivMMC автоматически подключает собственное ПЗУ по адресам \$0000–\$1FFF. Ловушка RST \$08 — основная точка входа API.

Паттерн API

Каждый вызов esxDOS следует одному и тому же паттерну:

```
db  function_id      ; function number (byte after RST)
; Returns:
;   Carry clear = success
;   Carry set   = error, A = error code
```

Номер функции — это байт, непосредственно следующий за инструкцией RST \$08 в памяти. Z80 выполняет RST \$08, что вызывает переход на адрес \$0008.

DivMMC автоматически подключает своё ПЗУ по этому адресу, читает следующий байт (номер функции), диспатчит вызов, затем отключает ПЗУ и возвращается к инструкции после DB.

Справочник функций

Функция	ID	Описание	Вход	Выход
M_GETSETDRV	\$89	Получить/установить диск по умолчанию	A = '*' для диска по умолчанию	A = буква диска
F_OPEN	\$9A	Открыть файл	IX = имя файла (нуль-терминированное), B = режим, A = диск	A = дескриптор файла (нуль-терминированное), A = дескриптор
F_CLOSE	\$9B	Закрыть файл	A = дескриптор файла	-
F_READ	\$9D	Прочитать байты	A = дескриптор, IX = буфер, BC = количество	BC = прочитано байт
F_WRITE	\$9E	Записать байты	A = дескриптор, IX = буфер, BC = количество	BC = записано байт
F_SEEK	\$9F	Позиционирование в файле	A = дескриптор, L = whence, BCDE = смещение	BCDE = новая позиция
F_FSTAT	\$A1	Статус файла (по дескриптору)	A = дескриптор, IX = буфер	11-байтный блок stat
F_OPENDIR	\$A3	Открыть каталог	IX = путь (нуль-терминированный)	A = дескриптор каталога
F_READDIR	\$A4	Прочитать запись каталога	A = дескриптор каталога, IX = буфер	запись по (IX)
F_CLOSEDIR	\$A5	Закрыть каталог	A = дескриптор каталога	-
F_GETCWD	\$A8	Получить текущий каталог	IX = буфер	путь по (IX)
F_CHDIR	\$A9	Сменить каталог	IX = путь	-

Функция	ID	Описание	Вход	Выход
F_STAT	\$AC	Статус файла (по имени)	IX = имя файла	11-байтный блок stat

Режимы открытия файла

Режим	Значение	Описание
Только чтение	\$01	Открыть существующий файл для чтения
Создать/обрезать	\$06	Создать новый или обрезать существующий для записи
Только создать	\$04	Создать новый файл; ошибка, если существует
Дополнение	\$0E	Открыть для записи в конец файла

Значения Seek Whence

Whence	Значение	Описание
SEEK_SET	\$00	Смещение от начала файла
SEEK_CUR	\$01	Смещение от текущей позиции
SEEK_END	\$02	Смещение от конца файла

Пример кода: загрузка файла

```

;
; 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

```

Пример кода: потоковое чтение данных из файла

Для демо, которые загружают данные инкрементально — распаковка фрагментов уровней между кадрами, потоковое воспроизведение предварительно отрендеренной анимации или подгрузка музыкальных паттернов по запросу — паттерн таков: открыть файл один раз, читать порцию за кадр, закрыть по завершении.

```
; 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

```

Обнаружение esxDOS

```

; 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

```

Более консервативный подход — проверить сигнатуру ловушки DivMMC перед вызовом любых функций API. На практике метод выше работает на всех моделях 128К, потому что обработчик \$0008 в ПЗУ 128К не падает — он выполняет безопасную последовательность и возвращается. На 48К-машине без esxDOS инструкция RST \$08 попадает в рестарт ошибок, для чего может потребоваться специальная обработка. Глава 21 обсуждает это в контексте продакшн-загрузчика игры.

3. +3DOS (Amstrad +3)

Amstrad Spectrum +3, со встроенным 3-дюймовым дисководом, имеет собственную ДОС: +3DOS. API использует другой механизм — вызовы точек входа в ПЗУ +3DOS на странице \$01, доступные через RST \$08 с другим набором кодов функций.

+3DOS редко используется в демосцене по двум причинам. Во-первых, +3 продавался преимущественно в Западной Европе и никогда не был доминирующей моделью Spectrum ни в одном демосцена-сообществе. Во-вторых, нестандартная раскладка памяти и схема переключения ПЗУ +3 делают его несовместимым с большинством демосценового кода, написанного для архитектуры 128K/Pentagon. Если тебе нужна совместимость с +3, API +3DOS документирован в техническом руководстве Spectrum +3 (Amstrad, 1987). Для большинства демо- и игровых проектов достаточно предоставить файл .tap — +3 загружает файлы .tap нативно через режим совместимости с лентой.

4. Практические паттерны

Загрузка экрана с диска (TR-DOS)

Загрузочный экран — первое впечатление пользователя. В TR-DOS файл экрана (SCREEN C, 6912 байт) загружается прямо по адресу \$4000 и появляется мгновенно:

```
; 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
```

Загрузка экрана с SD (esxDOS)

Тот же визуальный результат, другой API:

```
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

```

Двухрежимный загрузчик

Продакшн-загрузчик должен определить доступное хранилище и использовать его:

```

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

```

Потоковое чтение сжатых данных

Самый мощный паттерн комбинирует API хранилища со сжатием (Приложение C). Открой файл со сжатыми данными, читай порции в буфер каждый кадр, распаковывай в целевую область и продвигайся:

```

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

```

При 256 байтах за кадр и 50 кадрах/сек ты получаешь потоковую скорость 12,5 КБ/сек с SD-карты — достаточно для сжатой полноэкранной анимации. На TR-DOS прямое чтение секторов со скоростью один сектор за кадр даёт 12,8 КБ/сек (256 байт * 50 кадров/сек). Узкое место — скорость распаковки, а не ввод-вывод.

5. Справочник форматов файлов

Формат	Расширение	Применение	Примечания
Образ диска TR-DOS	.trd	Стандарт для релизов на Pentagon/Scorpion	Сырой образ 640 КБ. Поддерживается всеми эмуляторами.
Файловый контейнер TR-DOS	.scl	Проще, чем .trd	Содержит файлы без полной структуры диска. Хорош для распространения.
Образ ленты	.tap	Универсальный ленточный формат	Работает на любой модели Spectrum и в любом эмуляторе. Без файловой системы.
Расширенный образ ленты	.tzx	Лента с защитой от копирования / турбо-загрузчиками	Сохраняет точные тайминги ленты. Редко нужен для новых релизов.
Снапшот (48К/128К)	.sna	Быстрая загрузка, без файловой системы	Захватывает полное состояние машины. Не нужен код загрузчика.
Снапшот (сжатый)	.z80	Как .sna, но сжатый	Несколько версий; .z80 v3 поддерживает 128К.
Дистрибутив Next	.nex	Исполняемый файл ZX Spectrum Next	Самодостаточный бинарник с заголовком, описывающим раскладку банков памяти.

Выбор формата релиза: Для демосценового релиза предоставь как минимум два формата:

1. **.trd** для пользователей TR-DOS (российско-украинское сообщество, владельцы Pentagon/Scorpion и пользователи эмуляторов, предпочтита-

ющие образы дисков). Это формат по умолчанию для пати вроде Chaos Constructions и DiHalt.

2. **.tap** для всех остальных (реальное железо 128K с ленточным входом, пользователи DivMMC через загрузчик .tap, и все эмуляторы). sjasmplus умеет генерировать .tap напрямую с помощью директивы SAVETAP.

Если твоё демо достаточно маленькое (менее 48 КБ), снапшот .sna тоже отлично подходит — он загружается мгновенно без кода загрузчика.

6. См. также

- **Глава 15** — Анатомия железа: переключение банков памяти, порт \$7FFD, полная карта портов, рядом с которой существуют TR-DOS и esxDOS.
- **Глава 20** — Рабочий процесс демо: форматы релизов, правила подачи на пати, требования к .trd и .tap.
- **Глава 21** — Полноценная игра: продакшин-качество кода загрузки с ленты и esxDOS, двухрежимное определение, побанковая загрузка.
- **Приложение С** — Сжатие: какие компрессоры сочетать с потоковым вводом-выводом.
- **Приложение Е** — eZ80 / Agon Light 2: файловый API MOS на Agon, предоставляющий аналогичные файловые операции (`mos_fopen`, `mos_fread`, `mos_fclose`) через другой механизм (RST \$08 с кодами функций MOS в режиме ADL).

Источники: WD1793 datasheet (Western Digital, 1983); дизассемблирование TR-DOS v5.03 (различные авторы, public domain); документация API esxDOS (Wikipedia, zxe.io); спецификация аппаратуры DivMMC (Mario Prato / ByteDelight); Spectrum +3 Technical Manual (Amstrad, 1987); Introspec, «Loading and saving on the Spectrum» (Hype, 2016)

Приложение I: Bytebeat и AY-Beat — генеративный звук на Z80

«При 256 байтах bytebeat — твой единственный реалистичный вариант: для паттернового проигрывателя просто нет места.» – Глава 13

Это приложение охватывает формульную генерацию звука на ZX Spectrum — от оригинальной концепции PCM-bytebeat до AY-адаптированной техники, которая порождает структурированную, эволюционирующую музыку из горстки инструкций Z80. Глава 13 знакомит с AY-beat как инструментом sizecoding. Данное приложение — полный справочник: теория, формулы, маппинг регистров и полностью рабочий движок, который можно встроить в 256-байтное интро.

Тебе понадобится справочник по регистрам AY из Приложения G, открытый параллельно с этим приложением. Каждый номер регистра, упомянутый здесь (R0, R7, R8, R11, R13 и т.д.), документирован там с полными побитовыми раскладками и адресами портов.

1. Классический Bytebeat: традиция PCM

В 2011 году Ville-Matias Heikkila (Viznut) опубликовал открытие, которое уже ходило в подпольных программистских кругах: одно выражение на Си, вычисляемое по разу на сэмпл с инкрементирующимся счётчиком t , способно порождать сложную ритмическую музыку, если интерпретировать выход как 8-битный беззнаковый PCM на 8 кГц.

Основная идея:

```
putchar( f(t) ); // pipe to /dev/dsp at 8000 Hz
```

Функция $f(t)$ — это обычно односторочное выражение, построенное из побитовых операций, умножения и битовых сдвигов. Никаких осцилляторов, никаких огибающих, никаких таблиц нот — только целочисленная арифметика над счётчиком.

Знаменитые формулы

t*((t>>12|t>>8)&63&t>>4) — оригинал Viznut. Каскадные ритмические тона, циклически проходящие через соотношения высот, создавая эффект чего-то среднего между музыкальной шкатулкой и сломанным телефоном. $t>>12$ и $t>>8$ создают две частотно-делённые версии счётчика; $\&63$ ограничивает диапазон; $\&t>>4$ ритмически стробирует выход. Умножение на t создаёт фундаментальную развёртку высоты.

t*(t>>5|t>>8)>(t>>16) — эволюционирующие ритмические паттерны. Сдвиг вправо на $t>>16$ означает, что весь характер звука меняется каждые ~ 8 секунд (65536 сэмплов при 8 кГц). Каждая 8-секундная секция обладает различным динамическим диапазоном и характером.

(t*5&t>>7)|(t*3&t>>10) — две переплетённые мелодические линии. $t*5$ и $t*3$ создают два потока высоты на разных интервалах; AND со сдвинутыми счётчиками независимо стробирует их; OR объединяет. Результат звучит как две переплетающиеся мелодии, играющие одновременно.

Почему это работает

Побитовые операции над инкрементирующими счётчиком создают периодические структуры одновременно на множестве временных масштабов. Рассмотри битовый паттерн t при счёте:

- Бит 0 переключается каждый сэмпл (4000 Гц — как высота неслышим, но формирует волновую форму)
- Бит 7 переключается каждые 128 сэмплов ($\sim 62,5$ Гц — басовая территория)
- Бит 12 переключается каждые 4096 сэмплов ($\sim 1,95$ Гц — ритмический пульс)
- Бит 15 переключается каждые 32768 сэмплов ($\sim 0,24$ Гц — структурное изменение)

Сдвиг вправо $t>>n$ выбирает, какой временной масштаб доминирует. Операции AND создают паттерны совпадений — моменты, когда два временных масштаба выравниваются. Операции OR объединяют независимые паттерны. Умножение на малые константы создаёт гармонические соотношения (соотношения частот). 8-битное усечение выхода действует как естественный формирователь волновой формы, заворачивая значения обратно в диапазон и создавая дополнительные гармоники.

Результат самоподобен: звук обладает ритмической структурой на каждом масштабе, от отдельных циклов колебания до многосекундных фразовых структур. Именно это самоподобие делает bytebeat похожим на музыку, а не на шум — хотя никакие музыкальные знания в формулу не закладывались.

На Spectrum: тупик бипера

Биперный выход ZX Spectrum — это однобитный динамик, управляемый битом 4 порта \$FE. В принципе, можно запустить формулу bytebeat и вывести результат:

```
; 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
```

Это работает и производит звук. Но потребляет 100% процессора — Z80 не делает ничего, кроме вычисления сэмплов и переключения динамика. Никаких обновлений экрана, никаких визуальных эффектов, никакой обработки ввода. Частота дискретизации тоже неправильная (слишком высокая), а точное её управление требует тщательного подсчёта тактов (T-state) с подбивкой NOP-ами.

Для демо это тупик. Бипер — это однобитный выход, требующий постоянного внимания процессора. Настоящая адаптация bytebeat к Spectrum требует совершенно иного подхода.

2. AY-Beat: bytebeat, переосмысленный для тонового генератора

AY-3-8910 — это не ЦАП. Он не принимает амплитудные сэмплы. Это программируемый тоновый генератор: ты задаёшь ему частоту (как значение периода), громкость (0-15) и необязательные параметры шума и огибающей, а его внутренние осцилляторы автономно генерируют звук. Процессор свободен для другой работы.

Ключевая идея AY-beat: **заменить счётчик сэмплов счётчиком кадров, а PCM-выход — значениями регистров AY.**

Классический bytebeat вычисляет один амплитудный сэмпл на ~8000 Гц. AY-beat вычисляет периоды тона, громкости и параметры шума на 50 Гц — раз за видеокадр, по инструкции HALT. Осцилляторы AY берут на себя саму генерацию звука между кадрами.

Счётчик кадров t заменяет счётчик сэмплов. Формулы оперируют t , но производят значения регистров, а не сэмплы волновой формы. Там, где PCM-bytebeat имеет одну степень свободы (амплитуда), AY-beat располагает множеством: три независимых периода тона (по 12 бит каждый), три громкости (по 4 бита каждая), один период шума (5 бит) и 16-битный период огибающей с выбором формы.

Базовая архитектура AY-Beat

```
; 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

```

Это простейший AY-beat: один канал, одна формула тона, одна формула громкости, ~30 байт. Он производит циклическую развёртку, которая поднимается по высоте и затухает и нарастает — не музыка, но узнаваемо структурированный звук.

Что меняется по сравнению с PCM-bytebeat

Аспект	Классический (PCM)	AY-Beat
Частота обновления	~8000 Гц	50 Гц (частота кадров)
Выход	8-битная амплитуда	Период тона (12 бит), громкость (4 бита), шум (5 бит)
Каналы	1 (моно-динамик)	3 тона + 1 шум + огибающая
Нагрузка на процессор	100% (все такты)	~200-500 Т за кадр (~0,3%)

Аспект	Классический (PCM)	AY-Beat
Масштабирование формул	Мелкозернистое, быстрая эволюция	Крупнозернистое, нужны более широкие сдвиги
Генерация звука	Процессор вычисляет каждый сэмпл	Аппаратные осцилляторы AY работают автономно

Частота кадров 50 Гц означает, что формулы эволюционируют в 160 раз медленнее, чем при 8 кГц. Чтобы получить эквивалентную ритмическую плотность, используй более крупные множители и меньше сдвигов вправо. Формула, которая даёт приятный ритм при 8 кГц с $t \gg 12$ (период $\sim 0,5$ сек при 8 кГц), потребует примерно $t \gg 4$ при 50 Гц для аналогичного тайминга ($\sim 0,3$ сек между повторениями). Общее правило: раздели величины сдвигов из PC-bytebeat на ~ 7 (\log_2 от соотношения 160x) и подстрой на слух.

3. Дрон: огибающая + тон (режим E+T)

Здесь AY-beat становится по-настоящему интересным. Генератор огибающей AY автоматически циклически меняет громкость канала без какого-либо участия процессора. Установи регистр громкости канала в режим огибающей (бит 4 = 1, т.е. запиши \$10 в R8/R9/R10), и аппаратная часть берёт на себя модуляцию громкости на частоте огибающей, заданной R11-R12.

Результат — дрон: непрерывно эволюционирующий тембр, создаваемый взаимодействием тонового осциллятора и осциллятора огибающей. Процессорная стоимость поддержания этого дрона почти нулевая — нужно лишь обновлять период тона и период огибающей раз в кадр, а аппаратура делает всё остальное.

Рецепт дрона

1. Задай период тона по формуле — он определяет базовую высоту.
2. Задай период огибающей по другой формуле — он определяет скорость модуляции.
3. Установи форму огибающей в повторяющуюся волновую форму (формы \$08, \$0A, \$0C или \$0E).
4. Установи громкость канала в режим огибающей (\$10).
5. Аппаратура производит непрерывно эволюционирующий дрон с нулевой посэмпловой нагрузкой на процессор.

```
; 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

```

Красота режима Е+Т — в интерференции двух частот. Когда период огибающей близок к периоду тона, возникают эффекты амплитудной модуляции — громкость биений на разностной частоте, создающая колеблющийся, органоподобный тембр. Когда частота огибающей значительно ниже частоты тона, она действует как медленное трепетание. Когда значительно выше — попадаешь на территорию buzz-bass (см. Приложение G и Главу 11).

Развёртка периода огибающей при одновременном движении периода тона создаёт непрерывно эволюционирующие текстуры. Две формулы создают двумерное пространство параметров, которое звук исследует во времени. С правильной парой формул дрон никогда не повторяется в точности — он блуждает по тембральным вариациям, создавая амбиентный звуковой ландшафт из менее чем 30 байт кода.

Стоимость в байтах

Настройка режима Е+Т с формулой занимает приблизительно 15-25 байт. Для 256-байтного интро это даёт богатый, эволюционирующий дроновый звук фактически бесплатно — без покадрового вычисления громкости, без паттерновых данных, только два значения регистров, выведенных из простых формул. Аппаратура AY выполняет всю работу по осцилляции.

4. Шумовая перкуссия

Генератор шума AY (R6) производит псевдослучайный шум на программируемой частоте (0-31). Регистр микшера (R7) управляет тем, какие каналы получают шум. Включение и выключение шума ритмически, управляемое счётчиком кадров, создаёт перкуссионные паттерны.

Базовый паттерн бочки

```
; 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:
```

Характер перкуссии в зависимости от периода шума

Значение R6	Характер	Применение
0-3	Резкий щелчок, пробивной	Бочка, римшот
4-8	Чёткое шипение	Тело малого барабана
10-15	Широкий шум	Открытый хай-хэт
20-31	Низкий гул	Далёкий гром, эмбиент

Ритмическое разнообразие через битовые маски

Различные маски AND на счётчике кадров дают различную ритмическую плотность:

Маска	Период	Частота	Характер
AND \$03	Каждые 4 кадра	12,5 Гц	Скоростная очередь, хай-хэт
AND \$07	Каждые 8 кадров	6,25 Гц	Стандартная бочка
AND \$0F	Каждые 16 кадров	3,125 Гц	Половинный темп, разреженный
AND \$1F	Каждые 32 кадра	1,5625 Гц	Медленный пульс, вступление

Комбинируй две маски для полиритмии: проверяй $t \text{ AND } \$07$ для бочки, $t \text{ AND } \$03$ для хай-хэта. Это стоит около 10 дополнительных байт, но добавляет существенную ритмическую сложность.

Использование огибающей для затухания барабана

Вместо ручного затухания громкости каждый кадр используй генератор огибающей AY в однократном режиме. Установи R13 в форму \$00 (затухание до нуля, удержание), и аппаратура обработает спад громкости автоматически:

```

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:
```

Это экономит несколько байт, устранив код ручного затухания. Компромисс: генератор огибающей общий для всех каналов в режиме огибающей. Если канал А использует Е+Т-дрон, канал С не может независимо использовать огибающую для затухания барабана. Планируй распределение каналов соответственно.

5. Многоканальная гармония

AY имеет три независимых тоновых канала. AY-beat может вывести все три из одной формулы, используя битовую ротацию, создавая впечатление контрапункта практически из ничего.

Три голоса из одной формулы

```
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
```

Битовые ротации создают версии одного и того же паттерна со сдвигом по фазе. Каналы играют связанные, но смешённые мелодии — они следуют одному контуру, но приходят к каждой высоте в разные моменты. Это создаёт впечатление контрапункта: множество независимых голосов, разделяющих общую логику.

Почему ротация создаёт гармонию

RRCA — это *ротация*, а не сдвиг: биты, выпадающие снизу, заворачиваются наверх. Это означает, что три канала проходят один и тот же набор значений периода в том же порядке, но со сдвигом во времени. Сдвиг зависит от величины ротации:

- **RRCA x 2:** Канал «впереди» примерно на четверть цикла паттерна. Это часто создаёт интервалы, которые звучат как кварты или квинты — не точно настроенные, но гармонически связанные достаточно, чтобы быть приятными.
- **RRCA x 4:** Половина байта сдвига. Это стремится создавать октавоподобные отношения, поскольку бит 4, ротированный в бит 0, фактически вдвое уменьшает период при определённых фазовых выравниваниях.

Это не настоящие музыкальные интервалы. Это псевдогармонические отношения, созданные структурой двоичных чисел. Но слух снисходителен — если два тона разделяют большую часть своего битового паттерна, они звучат «родственno», и для 256-байтного интро этого достаточно.

Формулы громкости для многоканальности

Задай каждому каналу свою формулу громкости, чтобы не все три голоса оказывались на одном уровне одновременно:

```

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

```

Инвертированная громкость на канале С создаёт динамику «вопрос-ответ»: по мере нарастания одного голоса другой затухает. Это стоит 2 дополнительных

байта (`XOR $0F`), но существенно улучшает музыкальную текстуру.

6. Кулинарная книга формул

Следующие формулы были протестированы на AY при частоте кадров 50 Гц. «Байты» — это стоимость реализации на Z80 для вычисления формулы из значения, уже находящегося в регистре А (счётчик кадров). Маска периода определяет диапазон высоты.

Формулы периода тона

#	Формула	Реализация на Z80	Байты	Звук	Лучшее применение
1	<code>t AND \$3F</code>	<code>and \$3F</code>	2	Восходящая пила, цикл 1,28 сек	Простая развёртка
2	<code>t*3 AND \$3F</code>	<code>ld e,a : add a,a : add a,e : and \$3F</code>	5	Быстрая разворотка, широкие интервалы	Энергичный бас
3	<code>t XOR (t>>3)</code>	<code>ld e,a : rrca : rrca : rrca : xor e</code>	5	Хаотичный с периодической структурой	Шумовая текстура
4	<code>(t AND \$0F) XOR \$0F</code>	<code>and \$0F : xor \$0F</code>	4	Трэтугольная волна, разворотка туда-сюда	Мелодический лид

#	Формула	Реализация на Z80	Байты	Звук	Лучшее применение
5	$t * 5 \text{ AND } t >> 2$	<code>ld e,a : add a,a : add a,a : add a,e : ld d,a : ld a,e : rrc a : rrc a : and d</code>	10	Ритмическое стро-бирование	Перкуссионный
6	$(t+t >> 4) \text{ AND } \$1F$	<code>ld e,a : rrc a : rrc a : rrc a : rrc a : add a,e : and \\$1F</code>	6	Медленно модурированная раз-вёртка	Эволюционирующий дрон
7	$t \text{ AND } (t >> 3) \text{ AND } \$1F$	<code>ld e,a : rrc a : rrc a : rrc a : and e : and \\$1F</code>	6	Самоподобный, фрактальный ритм	Сложные паттерны
8	$(t >> 1) \text{ XOR } (t >> 3)$	<code>ld e,a : rrc a : ld d,a : rrc a : rrc a : xor d</code>	6	Двухскогорстная интерфренция	Металлическая текстура
9	$t * 7 \text{ AND } \$7F$	<code>ld e,a : add a,a : add a,a : sub e : and \\$7F</code>	6	Широкая раз-вёртка, 7x скость	Ощущение быстрого арпеджио
10	$(t \text{ XOR } t >> 1) \text{ AND } \$3F$	<code>ld e,a : rrc a : xor e : and \\$3F</code>	5	Последовательность кода Грэя	Ступенчатая мелодия

642ПРИЛОЖЕНИЕ I: BYTEBEAT И AY-BEAT — ГЕНЕРАТИВНЫЙ ЗВУК НА Z80

#	Формула	Реализация на Z80	Байты	Звук	Лучшее применение
11	$t \text{ AND } \$07$ OR $t >> 4$	<code>ld e,a : and \$07 : ld d,a : ld a,e : rrca : rrca : rrca : or d</code>	8	Вложен-ные цик-лы, два рит-миче-ских слоя	Многослойный ритм
12	$(t+t+t>>2)$ AND $\$3F$	<code>ld e,a : add a,a : ld d,a : ld a,e : rrca : rrca : add a,d : and \$3F</code>	8	Уско-ряю-щаяся раз-вёрт-ка с под-пат-тер-ном	Текстурированный лид

Формулы громкости

#	Формула	Z80	Байты	Эффект
V1	$t >> 3 \text{ AND } \$0F$	<code>rrca : rrca : rrca : and \$0F</code>	5	Медленный цикл затухания, 5,12 сек
V2	$(t \text{ AND } \$0F) \text{ XOR } \$0F$	<code>and \$0F : xor \$0F</code>	4	Треугольная громкость, туда-сюда
V3	$t * 3 >> 4 \text{ AND } \$0F$	<code>ld e,a : add a,a : add a,e : rrca : rrca : rrca : and \$0F</code>	8	Нерегулярный паттерн затухания
V4	$\$0F$ (константа)	<code>ld d,\$0F</code>	2	Максимальная громкость, для режима огибающей

Как читать таблицу

Выбери формулу тона и формулу громкости. Скомбинируй их. Общая стоимость в байтах — это сумма обеих реализаций плюс накладные расходы на запись регистров AY (~8 байт на канал для двух вызовов `ay_write`: выбор регистра +

данные для младшего байта тона и громкости). Один канал с формулой #1 и громкостью V1 стоит приблизительно $2 + 5 + 16 = 23$ байта, включая запись регистров.

Формула #10 (код Грея) заслуживает особого упоминания. Последовательность кода Грея меняет только один бит за шаг, поэтому период тона меняется ровно на одну единицу за кадр — плавная, ступенчатая мелодия. В сочетании с маской AND она циклически проходит ограниченный диапазон высот с приятной регулярностью. Это одна из наиболее музикально звучащих одиночных формул.

7. Собираем вместе: полный движок AY-Beat

Вот полный, минимальный движок AY-beat, производящий 3-канальный генеративный звук с дроном на огибающей. Это движок, который ты вставляешь в 256-байтное интро рядом с визуальным эффектом.

```
; 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.
; =====

```

Что это производит

- **Канал А:** Простая восходящая развёртка, циклически проходящая периоды 0-63 каждые 64 кадра (1,28 секунды). Фундаментальный паттерн.
- **Канал В:** Та же развёртка с 3-кратной скоростью, создающая быстро меняющиеся интервалы. Когда она совпадает с каналом А, ты слышишь консонанс; когда расходится — диссонанс. Чередование создаёт ритмический интерес.
- **Канал С:** Развёртка по коду Грея в режиме огибающей. Треугольная огибающая создаёт автоматическую модуляцию громкости, порождая дрон, который фазируется относительно периода тона. Это гармоническая подложка под двумя другими голосами.
- **В целом:** Эволюционирующая, самоподобная текстура, циклически проходящая тональные соотношения. Звучит чуждо и механистично — именно то, что нужно для 256-байтного интро.

Точки настройки

Измени формулы тона. Замени любую из последовательностей AND/RRCA на другую формулу из кулинарной книги (раздел 6). Каждая замена полностью меняет характер.

Добавь шумовую перкуссию. Вставь блок `ld a,e : and $07 : jr nz,.no_hit` (раздел 4), чтобы добавить ритмические удары. Стоимость: ~12 байт. Займи канал (обычно В) или наложи шум на канал С.

Используй пентатоническую маскировку. Вместо AND \$3F как финальной маски, индексируй в 5-байтную пентатоническую таблицу подстановки. Это ограничивает периоды тона гармонически связанными значениями, делая выход более преднамеренно музыкальным. Стоимость: ~8 байт (5 на таблицу, 3 на поиск). Глава 13 обсуждает эту технику.

Варьируй фиксированные громкости. Замени константные записи громкости на формулы громкости из раздела 6. Даже `ld a,e : rrca : rrca : rrca : and $0F` (5 байт на канал) добавляет значительный динамический интерес.

8. Продвинутое: комбинирование техник

Предыдущие разделы покрывают отдельные строительные блоки. Хорошо сделанный движок AY-beat комбинирует несколько:

Архитектура для 256-байтного интро

```

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)

```

Суммарная нагрузка на процессор за кадр составляет приблизительно 300-500 тактов (T-state) — значительно менее 1% от ~70000 тактов (T-state), доступных за кадр. Оставшиеся 99% доступны для визуального эффекта.

Бюджет регистров

AY имеет 14 записываемых регистров. В минимальном движке AY-beat ты обычно записываешь 8-10 за кадр:

Регистр	Запись	Источник
R0 (Tone A low)	Каждый кадр	Формула
R2 (Tone B low)	Каждый кадр	Формула
R4 (Tone C low)	Каждый кадр или однократно	Формула или фиксированный
R1, R3, R5 (Tone high)	Однократно (установить в 0)	Константа
R7 (Mixer)	Каждый кадр или однократно	Константа или переключается для шума
R8, R9 (Volume A, B)	Каждый кадр	Формула или константа
R10 (Volume C)	Однократно	\$10 (режим огибающей)
R11 (Envelope low)	Каждый кадр	Формула
R13 (Envelope shape)	Однократно (кадр 0)	Константа

Регистры, которые можно полностью пропустить: R6 (период шума — нужен только при использовании шума), R12 (старший байт огибающей — устанавливается однократно в 0 для коротких периодов), R14-R15 (порты ввода-вывода — не относятся к звуку).

Расклад по размеру

Для 256-байтного интро каждый байт на счету. Вот как выглядит типичный бюджет AY-beat:

Компонент	Байты
Подпрограмма записи AY	9
Управление счётчиком кадров	5
3 формулы тона (простые)	12-18
3 настройки громкости	6-15
Настройка микшера	5
Настройка огибающей	8-12

Компонент	Байты
Итого	45-64

Это оставляет 192-211 байт на визуальный эффект, основной цикл и любую другую инфраструктуру. При 45 байтах движок из раздела 7 близок к оптимуму по соотношению объёма производимого звука к размеру.

9. AY-как-ЦАП: классический bytebeat через регистр громкости

Есть промежуточный путь между тупиком бипера и переосмыслением AY-beat. Регистры громкости AY-3-8910 (регистры 8, 9, 10) принимают 4-битные значения (0-15). Если обновлять регистр громкости с высокой частотой — скажем, в тесном цикле — выход AY становится 4-битным ЦАП. Именно так работают оцифрованная речь и проигрывание сэмплов в демо для Spectrum.

В применении к bytebeat: вычисляем $f(t)$, сдвигаем вправо до 4 бит, записываем в регистр громкости:

```
; 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
and e            ; A = t >> 5 (using DE as 16-bit t)
                 ; 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

```

Это даёт узнаваемый bytebeat — настоящие формулы волновой формы из раздела 1, слышимые через AY. Качество звука лучше, чем у бипера (4-битное разрешение против 1-битного), а выходной каскад AY обеспечивает правильные уровни аудиосигнала.

Стоимость по-прежнему жёсткая: ~80% процессора. Остаётся тонкая полоска времени для визуалов — достаточно для медленно обновляемого атрибутного эффекта, недостаточно для чего-то амбициозного. Эта техника полезна, когда тебе нужен конкретный звук классических формул bytebeat и ты готов заплатить процессорную цену.

Три пути вывода в сравнении

Путь	Разрешение	Нагрузка на процессор	Характер звука	Практично для демо?
Бипер (порт \$FE)	1 бит	~100%	Резкий, жужжащий	Нет
AY volume DAC	4 бита	~80%	Классический bytebeat	Едва (только атрибутные эффекты)
AY-beat (регистры)	Тон/шум	~0,5%	Чиптюн, генеративный	Да — правильный выбор

Для size-coded интро и демо AY-beat — почти всегда верный выбор. AY-как-ЦАП оставь для арт-проектов, где конкретная звуковая эстетика bytebeat — это суть.

10. Теория музыки для алгоритмов

Формулы AY-beat, игнорирующие теорию музыки, производят интересный шум. Формулы, которые кодируют теорию музыки, производят настоящую музыку. Следующие техники добавляют музыкальность за минимальное число байт.

Таблицы гамм: ограничение выхода приятными нотами

Сырая формула вроде `tone = t AND $3F` производит все 64 возможных значения периода — большинство из которых не имеют музыкальной ценности. **Таблица гаммы** отображает выход формулы в реальные периоды нот, гарантируя, что каждое значение звучит хорошо.

Гамма	Ноты	Размер таблицы	Характер
Пентатоника	5 (C D E G A)	10 байт (5 x 2-байтных периодов)	Всегда консонантная, фолк/этно
Мажорная диатоника	7 (C D E F G A B)	14 байт	Яркая, западная, знакомая
Минорная диатоника	7 (C D Eb F G Ab Bb)	14 байт	Тёмная, меланхоличная
Блюзовая	6 (C Eb F F# G Bb)	12 байт	Грубая, экспрессивная
Хроматическая	12	24 байта	Атональная, диссонантная — обычно неверный выбор для sizecoding

Пентатоника — лучший друг сайзкодера: 5 нот, 10 байт, и любая комбинация нот звучит приемлемо. На пентатонической гамме невозможно сыграть неправильную ноту. Вот почему так много 256-байтных интро звучат смутно «по-азиатски» или «по-фолковому» — пентатоническое ограничение делает случайные последовательности музыкальными.

```
; 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
```

Октачная деривация: бесплатный диапазон высоты

Сохрани одну октаву периодов. Выводи все остальные битовым сдвигом:

- SRL D : RR E = одна октава вверх (период вдвое, высота удвоена)
- SLA E : RL D = одна октава вниз (период удвоен, высота вдвое)

Пять пентатонических нот x одна сохранённая октава x битовые сдвиги = 5 нот x 5+ октав = 25+ различных высот из 10 байт данных. Формула выбирает ноту, отдельная битовая маска выбирает октаву:

```
; octave = note_index / 5 (0-2)
; note = note_index % 5
; Look up base period, then SRL 'octave' times
```

Арпеджио: тоны аккорда последовательно

Арпеджио циклически проходит тоны аккорда. В терминах ступеней гаммы:

Аккорд	Смещения ступеней	Звучание
Мажорное трезвучие	0, 2, 4 (основной тон, терция, квинта)	Яркое, разрешённое
Минорное трезвучие	0, 2, 3 (основной тон, м. терция, квинта)	Тёмное, напряжённое
Пауэр-аккорд	0, 4 (основной тон, квинта)	Открытый, мощный
Сусpendирован-	0, 3, 4 (основной тон, квarta, квинта)	Неопределённый, парящий
ный		

Реализация: `arp_step = (t / speed) % chord_size`, затем прибавить смещение к текущему основному тону:

```

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)

```

Три байта на форму аккорда. Скорость арпеджио выводится из счётчика кадров — отдельный таймер не нужен.

Пошаговые орнаменты: трели, морденты и глиссандо

Орнамент — это крошечный циклический паттерн относительных смещений высоты, накладываемый на ноту. В трекерной музыке орнаменты оживляют плоские тона:

Орнамент	Паттерн	Эффект	Байты
Трель	0, +1, 0, -1	Быстрое чередование с соседом	4

Орнамент	Паттерн	Эффект	Байты
Мордент	0, +1, 0, 0	Краткий верхний сосед, затем устаканивание	4
Глиссандо вверх	0, 0, +1, +1	Постепенный подъём	4
Вибралто	0, +1, +1, 0, -1, -1	Плавное колебание	6

Применяй, прибавляя значение орнамента к индексу ноты перед обращением к таблице гаммы:

```

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

```

Четыре байта превращают статичный тон в живой голос. Накладывай разные орнаменты на разные каналы для богатой текстуры.

Аkkордовыe поcледовательности: гармоническое движение

Основной тон аккорда может меняться со временем, следуя прогрессии. Классическая гармония в 4 байтах:

```

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)

```

Четыре байта данных прогрессии, циклически управляемые счётчиком кадров, дают твоей AY-beat-пьесе гармоническое движение — ощущение, что она «куда-то идёт», а не зацикливается на одном аккорде. Другие прогрессии:

Прогрессия	Ступени	Байты	Ощущение
I-IV-V-I	0, 3, 4, 0	4	Классическое разрешение
I-V-vi-IV	0, 4, 5, 3	4	Поп/рок стандарт
i-VI-III-VII	0, 5, 2, 6	4	Эпический минор
I-I-I-I	0, 0, 0, 0	1 (или пропустить)	Дрон/медитативный

Суммарный бюджет данных для богатой музыки

Комбинируя все техники:

Компонент	Байты
Пентатоническая таблица (5 нот)	10
Паттерн арпеджио (1 аккорд)	3
Орнамент (трель)	4
Прогрессия (4 аккорда)	4
Итого	21

21 байт музыкальных данных — плюс ~45 байт кода движка — производит трёхканальную музыку с мелодией, гармонией, сменой аккордов и орнаментикой. Пример `aybeat.a80` в сопроводительном коде этой книги демонстрирует этот подход в 320 байтах, с запасом для визуалов.

11. L-системные грамматики: фрактальные мелодии

Системы Линденмайера (L-системы) — это переписывающие грамматики, изначально изобретённые для моделирования роста растений. В применении к музыке они генерируют самоподобные последовательности с дальнодействующей структурой из крошечных наборов правил.

Концепция

L-система имеет **аксиому** (начальную строку) и **правила порождения** (правила раскрытия). Каждая итерация заменяет каждый символ согласно его правилу:

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

Это **L-система Фибоначчи**. Последовательность растёт в соотношении Фибоначчи (~1,618x за шаг). Отобрази символы в музыкальные события:

Символ	Музыкальное значение
A	Играть основной тон (ступень гаммы 0)
B	Играть квинту (ступень гаммы 4)

Результирующая мелодия: основной тон, квинта, основной тон, основной тон, квинта, основной тон, квинта, основной тон... — последовательность, которая не периодична и не случайна, а *квазипериодична*. Она обладает структурой на каждом масштабе, как фрактал. Звучит преднамеренно, но без повторений.

Почему L-системы работают для музыки

- Самоподобие.** Мелодия на больших масштабах повторяет мелодию на малых масштабах. Именно это делает сочинённую музыку связной — темы повторяются на разных уровнях.
- Неповторимость.** В отличие от зацикленного паттерна, последовательность L-системы никогда точно не повторяется (для иррациональных коэффициентов роста). Она остаётся интересной.
- Крошечная кодировка.** Правила — это несколько байт. Генерируемая ими последовательность произвольно длинна.

Полезные правила L-систем

Название	Аксиома	Правила	Рост	Характер
Фибоначчи	A	A→AB, B→A	~1,618x	Квазипериодический, органичный
Туэ-Морс	A	A→AB, B→BA	2x	Сбалансированный, честный — без длинных серий
Удвоение периода	A	A→AB, B→AA	2x	Всё более синкопированный
Кантор	A	A→ABA, B→BBB	3x	Разреженный, с паузами (B=пауза)

Реализация на Z80

Трюк для Z80 — **не раскрывать строку в памяти** (это потребовало бы неограниченного буфера). Вместо этого вычисляй символ на позиции n рекурсивно: отслеживай назад по применением правил, чтобы определить, от какого исходного символа произошла позиция n.

Для L-системы Фибоначчи существует элегантный короткий путь. Символ на позиции n зависит от представления Цекендорфа (кодирования Фибоначчи) числа n. Но для практического sizecoding работает более простой подход:

```
; 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

```

Более практичный подход для sizecoding: предвычисли несколько итераций L-системы в короткий буфер при инициализации (одна итерация Фибоначчи из 5-символьной аксиомы даёт 8 символов, две итерации — 13, три — 21,

всё вмещается в маленький буфер), затем циклически проходи буфер как мелодическую последовательность:

```
; 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
```

Мелодия как движение, а не абсолютные ноты

Наиболее музыкальное применение L-систем — отображение символов не в фиксированные ноты, а в **направления шага по гамме**. Мелодия фундаментально — это *движение*: вверх, вниз, повтор, скачок — по гамме. Начальная нота произвольна; контур — вот что важно.

Определи символы как движения:

Символ	Значение	Шаг по гамме
U	Шаг вверх	+1
D	Шаг вниз	-1
R	Повтор	0
S	Скачок вверх (прыжок)	+2

Теперь L-система генерирует мелодический контур, а не фиксированные последовательности высот:

Axiom: U
Rules: U → U R D, D → U, R → U D

656ПРИЛОЖЕНИЕ I: BYTEBEAT И AY-BEAT — ГЕНЕРАТИВНЫЙ ЗВУК НА Z80

```
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  ...
```

Мелодия ходит вверх и вниз по текущей гамме, всегда оставаясь в пределах таблицы гаммы. Она естественно тяготеет к стартовой высоте (возвраты балансируют уходы), создавая дугу напряжения и разрешения, которая делает музыку осмысленной.

```
; 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
```

Это более музикально, чем отображение А=основной тон, В=квинта. Одни и те же правила L-системы порождают разные мелодии в зависимости от стартовой ноты и лежащей в основе гаммы — смени гамму с пентатоники на блюзовую, и тот же контур создаёт совершенно другое настроение.

Трибонауччи: три символа для более богатых паттернов

L-система Фибоначчи использует два символа. **Трибонауччи** использует три: А→ABC, В→A, С→B. Коэффициент роста ~1,839x (постоянная трибонауччи). Три символа означают более разнообразное мелодическое содержание:

Символ	Как движение	Как нота
A	Шаг вверх (+1)	Основной тон
B	Повтор (0)	Терция
C	Шаг вниз (-1)	Квинта

```
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
```

Последовательность трибонауччи имеет более длинные неповторяющиеся серии, чем Фибоначчи, и более сложную внутреннюю структуру. Музикально трёхсимвольный словарь даёт мелодиям больше разнообразия — они не просто прыгают туда-сюда между двумя состояниями.

Мелодии PRNG с отобранными сидами

Линейный регистр сдвига с обратной связью (LFSR) или аналогичный ГПСЧ генерирует детерминистическую псевдослучайную последовательность из

начального значения (сида). Последовательность звучит случайно, но точно повторяется при сбросе сида. Это даёт воспроизводимые мелодические фрагменты.

Техника: протестируй много сидов, оставь те, что звучат хорошо. Сохрани 2-4 значения сидов (по 2 байта каждый) для разных секций произведения. Во время выполнения загрузи сид и позволь ГПСЧ генерировать мелодию. Сам ГПСЧ — это ~6-8 байт; каждый сид — 2 байта.

```
; 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
```

Рабочий процесс: напиши тестовый стенд, который проигрывает мелодию ГПСЧ для каждого значения сида 0-65535, слушай (или анализируй), отмечай удачные. На практике несколько часов тестирования дают десятки пригодных сидов. Сохрани 3-4 из них и переключайся между секциями произведения.

Комбинирование с таблицами гамм: выход ГПСЧ проходит через пентатоническую таблицу, поэтому даже «плохие» сиды производят консонантные ноты. Ты отбираешь по мелодическому контуру, а не избегаешь неправильных нот — таблица гаммы уже позаботилась об этом.

Комбинирование с L-системами: используй ГПСЧ для выбора того, какое правило L-системы применить на каждом шаге, создавая стохастические L-системы. Сид управляет «характером» произведения; грамматические правила управляют структурой. Этот гибрид производит наиболее богатый результат из наименьшего числа байт.

Комбинирование L-систем с другими техниками

L-системы генерируют последовательности нот. Комбинируй с другими техниками из этого приложения:

- **Таблица гаммы** отображает символы L-системы в реальные периоды АУ
- **Орнаменты** добавляют выразительность каждой ноте
- **Арпеджио** превращает каждую ноту L-системы в аккорд
- **Дрон на огибающей** обеспечивает устойчивую гармоническую подложку под фрактальной мелодией

- **Аккордовая прогрессия** меняет основной тон — мелодия L-системы транспонируется к каждому аккорду

Результат: крошечная программа (~60-80 байт музыкального кода + 20 байт данных), генерирующая минуты структурно связной, неповторяющейся, гармонически обоснованной музыки. Это алгоритмическая композиция, а не случайный шум — и она вмещается в size-coded интро.

Другие грамматики для музыки

Помимо L-систем, другие формальные грамматики порождают интересные музыкальные последовательности:

Клеточные автоматы. Правило 30 или Правило 110, применённые к строке бит, порождают сложные паттерны. Отобрази позиции бит в события включения/выключения нот. Стоимость: ~15 байт на правило клеточного автомата, ~20 байт на пошаговый вычислитель.

Евклидовы ритмы. Распредели к ударов равномерно по n шагам. Этот алгоритм (связанный с евклидовым НОД) генерирует ритмические паттерны, встречающиеся в музыке по всему миру: 3-из-8 — это тресильо, 5-из-8 — синкильо, 7-из-12 — распространённый западноафриканский паттерн колокола. Реализация — ~20 байт, и она создаёт идеальные ритмические основы для любого движка AY-beat.

См. также

- **Глава 11** — архитектура AY-3-8910, теория тона/шума/огибающей, техника buzz-bass
 - **Глава 12** — интеграция музыкального движка, синхронизация с эффектами, гибридные цифровые барабаны
 - **Глава 13** — техники sizecoding, место AY-beat в размерных категориях 256b/512b/1K/4K
 - **Приложение G** — полный справочник регистров AY с побитовыми раскладками, адресами портов и таблицами нот
-

Источники: Viznut (Ville-Matias Heikkila), “Algorithmic symphonies from one line of code - how and why?” (2011); countercomplex.blogspot.com; Глава 13 этой книги; различные 256-байтные интро для ZX Spectrum с Pouet.net

Приложение J: Современные инструменты для создания ретро-демо

«Лучший инструмент — тот, который переносит данные из твоей головы в память Z80 за минимальное число шагов.»

J.1 Два мира, одна философия

Демосцена на PC и демосцена на Z80 выглядят как разные планеты. На одной планете кодеры пишут фрагментные шейдеры на GLSL, процедурно генерируют меши через графы нодов и сжимают всё в 64К-исполняемые файлы, рендерящие трёхмерную графику в реальном времени при 60fps на потребительских GPU. На другой планете кодеры вручную пишут подпрограммы на ассемблере Z80, умещая четырнадцать эффектов в 128К банкированной памяти, считая такты, чтобы каждый кадр уложился ровно в 71 680 тактов.

Разрыв в аппаратных возможностях — огромный. Разрыв в философии — почти нулевой.

Оба мира поклоняются ограничениям. 64К-интроверс на PC ограничено размером файла столь же безжалостно, как демо на ZX Spectrum ограничено скоростью процессора. Участник Shader Showdown, пишущийреймарчер за 25 минут, испытывает то же творческое давление, что и автор 256-байтной программы, оптимизирующий переиспользование регистров. Процедурная генерация — построение контента из алгоритмов вместо хранения его как данных — является центральной техникой на обеих платформах, потому что обе наказывают за данные и поощряют вычисления (PC наказывает хранимые данные через ограничения размера файла; Spectrum наказывает их через ограничения памяти).

В этом приложении рассматривается современный инструментарий демосцены: редакторы синхронизации, генераторы данных, музыкальные синтезаторы, упаковщики исполняемых файлов и шейдерные инструменты. Большинство этих инструментов рассчитаны на x86/GPU и не могут работать на Z80 напрямую. Но они выполняют три роли в создании демо для ZX Spectrum:

1. **Генераторы данных** — вычисляют траектории, позиции частиц, процедурные текстуры на PC, затем экспортят результаты как сжатые бинарные таблицы, которые Z80 воспроизводит.

2. **Планировщики синхронизации** — проектируют временную связь между музыкой и визуалами в интерактивном редакторе, затем экспортируют номера кадров и кривые параметров в таблицы Z80 dw.
3. **Среды прототипирования** — тестируют визуальные алгоритмы на полной скорости на современном GPU, затем переводят работающий алгоритм на ассемблер Z80, точно зная, как должен выглядеть целевой результат.

Философия последовательна: **используй любой инструмент для подготовки, но рантайм Z80 — это рукописный ассемблер.** Качество демо оценивается по тому, что работает на Spectrum, а не по тому, что работает на PC разработчика. Инструменты — это леса; здание — это Z80.

J.2 Инструменты синхронизации

Проблема синхронизации

Синхронизация — попадание нужного визуального события в нужный музыкальный момент — это самая сложная часть создания демо (Глава 20). На уровне Z80 синхронизация — это всегда таблица данных: номера кадров в паре с действиями. Вопрос в том, как определить эти номера кадров эффективно и быстро итерировать.

Пять инструментов решают эту проблему, каждый со своим рабочим процессом.

GNU Rocket

Что это. GNU Rocket — это синк-трекер — редактор в стиле трекера, где столбцы представляют именованные параметры (`camera:x`, `fade:alpha`, `effect:id`), а строки — временные шаги (обычно кадры или музыкальные доли). Ты ставишь ключевые кадры на определённых строках и выбираешь режим интерполяции между ними: **step** (мгновенная смена), **linear** (постоянная скорость), **smooth** (кубическое сглаживание), или **ramp** (экспоненциальная). Демо подключается к редактору Rocket по TCP в процессе разработки. Ты перематываешь таймлайн, редактируешь значения и видишь обновление демо в реальном времени.

Кто использует. Rocket — де-факто стандартный инструмент синхронизации на PC и Amiga демосценах. Logicoma, Noice, Loonies, Adapt и десятки других групп используют его. Он был портирован на C, C++, C#, Python, Rust и JavaScript.

Рабочий процесс для Z80. ZX Spectrum не может запустить TCP-клиент, но концепция переносится напрямую:

1. Проектируй синк-треки в Rocket на PC, перематывая под музыку
2. Экспортируй данные ключевых кадров как бинарный файл (нативный экспорт Rocket)
3. Запусти Python-конвертер: квантуй float в 8-битные или 16-битные значения с фиксированной точкой, выдавая таблицы db/dw
4. Подключи таблицы в демо через INCBIN

Код Z80 просто читает таблицу — никакого TCP, никаких float, никакой сложности. Ты получаешь интерактивный опыт редактирования Rocket при разработке и минимальные накладные расходы рантайма Spectrum в финальном бинарнике.

Интерполяция. Четыре режима интерполяции Rocket чисто отображаются на воспроизведение Z80: - **Step** → просто используй значение напрямую (0 тактов на интерполяцию) - **Linear** → предвычисли дельту на кадр, прибавляй каждый кадр (~20 T-states) - **Smooth/Ramp** → запеки интерполированную кривую в экспортированную таблицу (Z80 читает предвычисленные значения, никакой интерполяции в рантайме)

Для большинства демо на ZX Spectrum запекание всех кривых в покадровые значения — самый простой подход. Демо на 3 000 кадров (одна минута при 50fps) с 4 параметрами синхронизации потребляет 12 КБ несжатых данных — существенно, но сжимаемо до 2-4 КБ с помощью ZX0.

Исходники: github.com/rocket/rocket (MIT-подобная лицензия)

FIGURE: GNU Rocket sync editor

Tracker-like grid with named columns:
 [camera:x] [camera:y] [fade:alpha] [effect:id]

Rows = frames/time steps. Keyframes shown as bright cells.
 Between keyframes: interpolation curves (step/linear/smooth/ramp)
 visualised as lines connecting values.

Bottom: transport controls (play, pause, scrub).
 Connected to running demo via TCP – edit live.

Screenshot needed: build GNU Rocket from source, create example
 project with 4 tracks, capture at a point showing all 4
 interpolation modes.

Vortex Tracker II

Что это. Vortex Tracker II — стандартный редактор ProTracker 3 для сцены ZX Spectrum. Ключевая функция для работы с синхронизацией — **счётчик кадров** в правом нижнем углу окна — он показывает абсолютный счёт прерываний (номер кадра) на текущей позиции воспроизведения.

Рабочий процесс синхронизации. Воспроизведи .pt3-файл. Следи за счётчиком кадров. Когда слышишь удар, акцент или переход фразы, к которому хочешь привязаться, запиши номер кадра. Внеси его в свою таблицу синхронизации. Пересобери демо, проверь, подкорректируй.

Kolnogorov (Vein) описывает это как свой основной метод: «Vortex + видеоредактор. В Vortex кадр показан в правом нижнем углу — я смотрел, к каким кадрам привязываться, создавал таблицу с записями dw frame, action и синхронизировал по ней.»

Форк VTI. Сообщество поддерживает VTI, форк Vortex Tracker II с дополнительными возможностями, включая улучшенную точность воспроизведения и

расширенную поддержку форматов. Для работы с синхронизацией оригинальный VT2 и VTI эквивалентны — счётчик кадров работает одинаково.

Ограничение. Итерирование медленное. Каждое изменение тайминга требует пересборки демо и просмотра с начала. Для простых демо с дюжины точек синхронизации этого достаточно. Для сложных демо с сотнями событий более интерактивный инструмент (Rocket, Blender) стоит затрат настройку.

FIGURE: Vortex Tracker II – frame counter

VT2 main window with pattern editor visible.
Bottom-right: position display showing pattern:row and absolute frame number.
Highlight/circle the frame counter.

Caption: "The frame number in VT2's status bar maps directly to the PT3 player's interrupt counter on the Spectrum. What you see here is what your sync table references."

Screenshot needed: open any .pt3 in VTI fork, play to a mid-song position, capture with frame number visible.

Blender VSE (Video Sequence Editor)

Что это. Встроенный нелинейный видеоредактор Blender. Для синхронизации демо он предоставляет таймлайн, на котором можно размещать цветные полосы (по одной на эффект), импортировать музыкальный трек как аудиополосу с видимой волновой формой и расставлять маркеры в точках синхронизации.

Рабочий процесс синхронизации: 1. Захвати каждый эффект, работающий в эмуляторе, как короткий видеокlip (или используй цветные полосы-заглушки) 2. Импортируй клипы и музыку (.wav) в VSE 3. Расположи клипы на таймлайне, перематывай, чтобы найти идеальные точки монтажа 4. Расставь маркеры (зелёные ромбики) на каждом событии синхронизации 5. Считай номера кадров с позиций маркеров

Визуальный рабочий процесс мощен: ты *видишь* волновую форму аудио, *видишь*, где попадает удар, и перетаскиваешь маркер в нужное место. Никаких вычислений, никаких конвертаций BPM в кадры.

Экспорт. Маркеры можно экспортить через Python-консоль Blender:

```
print(f"dw {m.frame}, 0 ; {m.name}")
```

Это выдаёт таблицу синхронизации, готовую для Z80, напрямую. Переименуй маркеры в соответствии с идентификаторами эффектов (plasma, flash, scroll) — и экспорт станет полноценной таблицей сцен.

DaVinci Resolve предоставляет аналогичные возможности (таймлайн + маркеры + экспорт EDL/CSV), и его бесплатной версии достаточно. Выбирай тот, который уже знаешь.

FIGURE: Blender VSE – demo storyboard

Timeline with 4-5 colour-coded strips (one per effect):

```
[TORUS: blue] [PLASMA: green] [DOTSCROLL: yellow] [ROTOZOOM: red]

Below: audio waveform strip (music.wav)
Vertical markers (green diamonds) at sync points.
Playhead at a transition point between effects.

Screenshot needed: create Blender project with dummy strips +
real AY music exported as WAV. Place ~8 markers at beat hits.
```

Blender Graph Editor

Что это. Редактор кривых Blender для анимированных свойств с ключевыми кадрами. Для работы над демо ты создаёшь пользовательские свойства объекта (например, scroll_speed, fade_alpha, camera_z) и ставишь на них ключевые кадры в соответствии с энергетикой и фазировкой музыки.

Почему это важно. Graph Editor даёт визуальный интерактивный контроль над тем, как параметры изменяются во времени — то же самое, что предоставляет GNU Rocket, но интегрированное в экосистему Blender. Ты видишь несколько кривых одновременно, корректируешь тайминг ключевых кадров перетаскиванием и переключаешь режимы интерполяции (constant, linear, Bezier) для каждого ключевого кадра.

Экспорт через Python API:

```
name = fcurve.data_path.split(' ')[1]
print(f"; {name}")
for kf in fcurve.keyframe_points:
    print(f"    dw {int(kf.co.x)}, {int(kf.co.y)}")
```

Это выдаёт данные синхронизации, готовые для Z80, напрямую. Проект Blender становится одновременно раскадровкой, справкой по синхронизации и конвейером данных — всё в одном файле.

FIGURE: Blender Graph Editor – keyframe export

Graph with X = frame number, Y = parameter value.
3 curves: scroll_speed (smooth ease-in), fade_alpha (step at transitions), camera_z (linear ramp).

Annotation showing the Python export:
for kf in fcurve.keyframe_points:
 print(f"dw {int(kf.co.x)}, {int(kf.co.y)}")

Arrow pointing to resulting Z80 data:
dw 0, 0 / dw 50, 128 / dw 150, 255 / ...

Screenshot needed: same Blender project, switch to Graph Editor view with 3 animated custom properties.

Motion Canvas

Краткое упоминание: **Motion Canvas** — это новый инструмент на TypeScript с MIT-лицензией для программного создания параметрических анимаций. Он спроектирован для объясняющих видео, но его подход «таймлайн как код» может служить средством планирования синхронизации для кодеров, предпочитающих писать код, а не перетаскивать ключевые кадры. Проект ещё на ранней стадии; следи за ним на motioncanvas.io.

Сравнительная таблица

Инструмент	Лицензия	Метод синхронизации	Путь экспорта для Z80	Интерактивный?
GNU Rocket	MIT-подобная	Треки с ключевыми кадрами + интерполяция	Бинарный → Python → таблицы dw	Да (TCP live editing)
Vortex Tracker II	Freeware	Считывание счётчика кадров	Вручную (записать номера кадров)	Частично (слушать + записывать)
Blender VSE	GPL	Маркеры на таймлайне + волновая форма	Python → таблицы dw	Да (визуальная перемотка)
Blender Graph Editor	GPL	Кривые с ключевыми кадрами	Python API → таблицы dw	Да (визуальное редактирование кривых)
DaVinci Resolve	Free/Commercial	Маркеры на таймлайне	EDL/CSV → Python → таблицы dw	Да (визуальная перемотка)
Motion Canvas	MIT	Таймлайн, определённый кодом	Экспорт из TypeScript (custom)	Программный

J.3 Предвизуализация и раскадровка

Прежде чем писать хоть одну строку ассемблера Z80, тебе нужно знать, как демо выглядит. Не в деталях — не каждый выбор палитры или скорость скроллинга — а общую структуру: какие эффекты, в каком порядке, какой длительности, с какими переходами. Это предвизуализация, и она экономит больше времени, чем любая оптимизация кода.

Blender как инструмент раскадровки

Grease Pencil в Blender (теперь полностью интегрированный как система 2D-рисования) позволяет набрасывать грубые представления каждого эффекта — цветные прямоугольники, простые формы, нарисованные от руки приближения.

Они не обязаны выглядеть как финальные эффекты. Они должны передавать: «здесь, в течение стольких-то секунд, происходит что-то голубое и кручёное».

Практический подход: 1. Создай один объект Grease Pencil на каждый эффект (цветной прямоугольник с названием эффекта) 2. Расположи их на таймлайне VSE вместе с музыкальным треком 3. Воспроизведи и посмотри на структуру

Ты делаешь не анимацию. Ты делаешь *расписание* — визуальное представление того, какой эффект когда работает и как долго. Цвета помогают заметить структурные проблемы: три медленных голубых эффекта подряд означают, что темп неправильный. 45-секундный отрезок без перехода означает, что зритель заскучает.

Видеоредакторы для черновой сборки

Когда у тебя есть эффекты, работающие в эмуляторе, захвати их как видеоклипы (OBS Studio, запись экрана или функции записи эмулятора). Импортируй клипы в видеоредактор — DaVinci Resolve, Blender VSE или даже iMovie — и расположи на таймлайне с музыкой.

Черновая сборка выявляет проблемы, невидимые в коде: - Эффект, который впечатляет по отдельности, но тянетя слишком долго в контексте - Переход, который должен попадать в удар, но оказывается между ударами - Два последовательных эффекта с похожими цветовыми палитрами, которые сливаются - Фрагмент, где визуальная энергия падает, в то время как музыка нарастает

Черновая сборка стоит часа работы. Она экономит дни корректировок тайминга на уровне ассемблера.

Рабочий процесс GABBA

Подход diver4d для GABBA (2019) пошёл дальше: он использовал **Luma Fusion**, iOS-videоредактор, в качестве основного инструмента планирования синхронизации. Рабочий процесс:

1. Закодируй каждый эффект по отдельности, протестируй в эмуляторе
2. Запиши экран с работающим эффектом
3. Импортируй записи + музыку в Luma Fusion на iPad
4. Расположи на таймлайне, перематывай покадрово, чтобы найти точки синхронизации
5. Запиши номера кадров, составь таблицу синхронизации

Ключевое понимание: покадровая синхронизация — это задача видеомонтажа, а не программирования. Решая её в инструменте, предназначенном для видеомонтажа, diver4d мог итерировать тайминг за секунды вместо минут. Код Z80 был слоем реализации; творческие решения принимались в видеоредакторе.

Kolnogorov о планировании синхронизации

Kolnogorov (Vein) формулирует комбинированный подход: «Я экспорттировал клипы с эффектами в видео, собирая их в видеоредакторе, прикрепляя музыкальный трек и смотрел, в каком порядке эффекты работают лучше всего, отмечая кадры, где должны происходить события.»

Важное слово — *смотрел*. Это визуальный, интуитивный процесс. Ты *видишь*, где удар попадает на волновую форму. Ты *видишь*, где переход эффекта ощущается правильно. И считаешь номер кадра. Никаких вычислений, никаких конвертаций ВРМ в кадры.

J.4 Генераторы данных

Демо на ZX Spectrum воспроизводит предвычисленные данные при 50fps. Вопрос: где вычислять эти данные? Для простых таблиц синусов и координат Брезенхэма достаточно Python-скрипта. Для сложных 3D-траекторий, органического движения частиц или жестов, захваченных в VR, нужна более мощная среда.

Unity как генератор данных

Unity — это не overkill для демо на ZX Spectrum: это overkill в качестве *движка демо*, но идеально в качестве *генератора данных*. Разница важна.

Захват движения в VR. XR Toolkit в Unity захватывает позицию VR-контроллера с частотой 90 Гц. Нарисуй траекторию в воздухе VR-контроллером — органическое движение руки невозможно воспроизвести математическими формулами. Понизь частоту до 50fps, квантий до 8-битных знаковых значений, примени дельта-кодирование, сожми. 5-секундная нарисованная вручную траектория превращается в 250 байт упакованных данных, которые *выглядят живыми* на Spectrum.

Системы GPU-частиц. VFX Graph в Unity запускает миллионы частиц на GPU. Прототипируй фонтан частиц, вихрь или симуляцию стаи, затем экспортируй позиции частиц по кадрам как CSV. На Spectrum ты рисуешь эти позиции как точки или ячейки атрибутов. Физическая симуляция, которая потребовала бы месяцев реализации на Z80, работает за миллисекунды на GPU.

Прототипирование шейдеров. Напиши плазму, тоннель или ротозумер как фрагментный шейдер в ShaderGraph Unity. Итерируй в реальном времени на полном разрешении, пока не будет выглядеть правильно. Затем переведи алгоритм на Z80, точно зная, как должен выглядеть целевой результат.

Unreal Engine как генератор данных

Unreal предлагает эквивалентные возможности через другие инструменты: - **OpenXR** для захвата движения в VR - **Niagara** для систем GPU-частиц - **Material Editor** для прототипирования шейдеров

Выбор между Unity и Unreal для генерации данных — вопрос знакомства. Оба экспортируют в одинаковые форматы (CSV, бинарные массивы). Оба предоставляют больше вычислительной мощности, чем тебе когда-либо понадобится для генерации данных Z80-демо.

Blender как генератор данных

Для большинства задач генерации данных Blender достаточно и полностью свободен:

- **Geometry Nodes** для процедурных траекторий — определи путь как сплайн, разбросай точки вдоль него, анимируй шумом, экспортируй позиции вершин по кадрам
- **Grease Pencil** для рисованной анимации — рисуй кадры, экспортируй координаты точек
- **Python API** для прямого экспорта — доступ к любому анимированному свойству из `bpy.data`
- **Физическая симуляция** для частиц, ткани, жидкости — симулируй, запеки, экспортируй покадровые данные

Blender не умеет захватывать движение в VR (без сторонних аддонов) и его система частиц работает на CPU (медленнее GPU-систем для миллионов частиц). Во всём остальном он не уступает Unity/Unreal и даже превосходит их для целей генерации данных.

Конвейер экспорта

Независимо от того, какой инструмент генерирует данные, конвейер до Z80 проходит одни и те же этапы:

- Export: float arrays (CSV, JSON, or binary)
- Python: float → 8-bit fixed-point (or 16-bit where needed)
- Delta-encode: store differences between frames (smaller values)
- Transpose: column-major layout (all X values, then all Y values)
- packbench analyze: verify entropy, check suggested transforms
- zx0/pletter: compress
- sjasmplus: INCBIN into demo

Каждый этап уменьшает размер данных:

Этап	Пример размера (250 кадров x 4 параметра)
Float CSV	~10 КБ
8-битная фиксируемая точка	1 000 байт
Дельта-кодирование	1 000 байт (значения уменьшаются, лучше степень сжатия)
Транспонирование + сжатие	300-500 байт

Этапы дельта-кодирования и транспонирования не уменьшают сырой размер — они переформатируют данные для лучшего сжатия. Столбцевая раскладка группирует похожие значения вместе (все дельты X, затем все дельты Y), что сжимается радикально лучше, чем построчная раскладка, где дельты X и Y чередуются.

Когда использовать что

Задача	Инструмент	Почему
Таблицы синусов, таблицы подстановки	Python-скрипт	Проще всего, без зависимостей
Процедурные 3D-траектории	Blender (Geometry Nodes)	Бесплатный, визуальный, Python-экспорт
Нарисованные вручную траектории анимации	Blender (Grease Pencil)	Рисуй прямо, покадрово
Захват жестов в VR	Unity (XR Toolkit) или Unreal (OpenXR)	Нужно VR-оборудование + рантайм
Позиции GPU-частиц	Unity (VFX Graph) или Unreal (Niagara)	Миллионы частиц, быстро
Прототип шейдерного алгоритма	Любой (ShaderToy, Unity, Unreal, Blender)	Какой знаешь

Предвычисленные векторные анимации Kolnogorov — подходящий пример: 3D-геометрия была вычислена оффлайн (минуты вычислений допустимы для 4К-интрано), полученные траектории вершин были сохранены как сжатые таблицы, и Spectrum воспроизводил их при 50fps. Инструмент, сгенерировавший траектории, не имеет значения для зрителя. Важно, что данные существуют и Z80 их воспроизводит.

J.5 Инструментарий РС-демосцены: краткая история

РС-демосцена потратила двадцать пять лет на создание инструментов для процедурной генерации контента и экстремального сжатия. Эти инструменты не могут работать на Z80, но их философия проектирования — процедурность во всём, маленько — это красивое, ограничение как творчество — в точности отражает то, что кодеры ZX Spectrum делают вручную. Понимание инструментария РС-сцены поможет тебе увидеть, какие проблемы уже решены (в других контекстах) и какие идеи можно адаптировать.

Farbrausch (1999-2012)

Farbrausch — немецкая демогруппа, переопределившая границы возможного в 64К- и 4К-исполняемых файлах. Их подход: строить инструменты, генерирующие всё процедурно, а затем упаковывать генераторы в крохотные исполняемые файлы.

Werkzeug (версии с 1 по 4) был их флагманским инструментом — нодовой процедурной системой, где текстуры, меши, анимации и композиции определялись как графы операторов. Никакие растровые изображения не хранились; каждый пиксель вычислялся в рантайме из рецепта математических операций. Инструмент использовался внутри Farbrausch; зрители видели лишь результатирующие исполняемые файлы.

Заметные работы, созданные с помощью Werkkzeug: - **fr-08: .the .product** (2000) — музыкальное 3D-видео в реальном времени в 64 КБ, переопределившее категорию 64К-интровера. Победило в компо демо на The Party 2000. - **.kkrieger** (2004) — шутер от первого лица в 97 280 байт. Текстуры, мешки, анимации, AI и звук — всё сгенерировано процедурно. Исполняемый файл меньше исходника этого приложения. - **fr-041: debris.** (2007) — кинематографическое демо в 177 КБ, продвинувшее качество рендеринга в реальном времени за пределы того, чего достигали многие полноразмерные демо. Победило на Breakpoint 2007.

kkrunchy — их упаковщик исполняемых файлов для 64К-интровера — контекстно-смесевой компрессор, достигавший степени сжатия далеко за пределами стандартных упаковщиков вроде UPX. kkrunchy до сих пор активно используется кодерами 64К-интровера, более чем десятилетие после роспуска Farbrausch.

V2 Synthesizer — программный синтезатор, спроектированный для интровера — VSTi-плагин для сочинения музыки с крохотным рантайм-проигрывателем, помещавшимся внутри упакованного исполняемого файла. Музыка хранилась как данные нот и параметры синтеза, а не как аудио.

В 2012 году Farbrausch выложили весь свой инструментарий в открытый доступ под BSD-подобной лицензией: github.com/farbrausch/fr_public. Репозиторий включает Werkkzeug (все версии), kkrunchy, V2 и ряд других инструментов. Код — капсула времени инженерии демосцены начала 2000-х: плотный C++, Win32 API, Direct3D 9 и креативные решения задач, которые современные GPU решают грубой силой.

Актуальность для Z80. Эквивалент процедурных текстур Werkkzeug на ZX Spectrum — твой Python-скрипт сборки, генерирующий таблицы подстановки из математических функций. Эквивалент kkrunchy — ZX0 или Pletter. Эквивалент V2 — AY-beat или проигрыватель Shiru для AY. Разный масштаб, тот же принцип: генерируй контент из компактных описаний, сжимай результат, распаковывай в рантайме.

TiXL (2024-наши дни)

TiXL (ранее Tool3) — духовный наследник Werkkzeug, разрабатываемый **Still** (pixtur/Thomas Mann) — демосценером, работавшим с участниками Farbrausch и продвинувшим нодовый процедурный подход дальше.

TiXL — среда для создания моушн-графики в реальном времени, построенная на современных GPU API. Как и Werkkzeug, она использует граф нодов для определения процедурного контента, но с двадцатью годами эволюции GPU за спиной: вычислительные шейдеры, физически корректный рендеринг, GPU-частицы и реймарчинг в реальном времени.

Распространяется под лицензией MIT (github.com/tixl3d/tixl); TiXL показывает, куда эволюционировала философия Werkkzeug. Концепция графа нодов — определение контента как рецепта операций, а не хранение его как данных — напрямую применима к разработке демо для Z80, даже при том что конкретные операции совершенно другие.

FIGURE: TiXL node graph

Visual programming canvas with connected nodes:
 [Time] → [Sine] → [Multiply] → [SetFloat]

A procedural texture pipeline:
[Noise3D] → [Remap] → [ColorGrade] → [RenderTarget]

3D scene graph:
[Mesh:Torus] → [Transform] → [Material] → [DrawMesh]
[Camera] → [Render] → [PostFX:Bloom] → [Output]

Right panel: live preview of the rendered output.
Bottom: timeline with playback controls.

Caption: "TiXL (MIT, 2024) carries forward the Werkzeug philosophy of node-based procedural content generation. The Z80 equivalent is a Python build script that generates lookup tables and compressed data from mathematical functions."

Screenshot needed: install TiXL, open an example project, capture the node graph + preview in a split view.

Bonzomatic

Bonzomatic (автор Gargaj / Conspiracy) — стандартный инструмент для соревнований **Shader Showdown** — живых шейдерных баттлов на демопати. Два кодера делят сцену, каждый пишет фрагментный шейдер с нуля за 25 минут, код и результат проецируются бок о бок для зрителей.

Bonzomatic предоставляет минималистичный редактор с окном предпросмотра в реальном времени. Каждое нажатие клавиши перекомпилирует шейдер; результат обновляется мгновенно. Нет сохранения, нет истории отмен за пределами буфера редактора, нет библиотек. Это чистое творчество в условиях ограничений — тот же дух, что и 256-байтное кодирование на Spectrum.

Исходники: github.com/Gargaj/Bonzomatic

Crinkler

Crinkler (авторы Rune Stubbe и Aske Simon Christensen / Loonies) — **сжимающий линкер** для Windows-исполняемых файлов. Там, где обычный линкер создаёт .exe, а ты сжимаешь его отдельно, Crinkler объединяет линковку и сжатие в один шаг, достигая степени сжатия, недоступной ни одному отдельному упаковщику.

Crinkler — стандарт для 1К, 4К и 8К PC-интров. Его сжатие настолько эффективно, что кодеры регулярно измеряют свою работу в «Crinkler-байтах» — финальный упакованный размер, который может составлять 60–70% от несжатого кода + данных.

Параллель с Z80: ZX0 и Pletter выполняют ту же роль для демо на Spectrum, хотя степень сжатия скромнее (код Z80 содержит меньше избыточности, чем x86).

Исходники: github.com/runestubbe/Crinkler (лицензия zlib)

Squishy

Squishy (авторы Logicoma) — упаковщик 64К-исполняемых файлов — эквивалент kkrunchy для современных 64К-интров. В отличие от Crinkler (который работает с крохотными интров), Squishy нацелен на размерный класс 64 КБ, где исполняемый файл содержит значительный объём шейдерного кода, процедур генерации текстур и музыки.

Распространяется только в бинарном виде; исходники недоступны. Упоминается здесь, потому что 64К-интров Logicoma (Happy Coding, Elysian, H - Immersion) представляют современное состояние искусства в категории 64К, и Squishy — часть этого конвейера.

Shader Minifier

Shader Minifier (автор Ctrl-Alt-Test) — минификатор GLSL/HLSL, который переименовывает переменные, удаляет пробелы и оптимизирует шейдерный код для минимального размера. Используется в sizecoding-интров, где важен каждый байт шейдерного исходника (шейдеры часто хранятся как строки в исполняемом файле и компилируются в рантайме).

Исходники: github.com/laurentlb/shader-minifier

z80-optimizer

z80-optimizer (автор oisee, 2025) — брутфорс-супероптимизатор Z80, написанный на Go. Он перебирает все пары инструкций Z80 (406 x 406 опкодов), тестирует каждую пару на всех возможных состояниях регистров и флагов и сообщает, когда более короткая или быстрая замена даёт идентичный результат. Никаких эвристик, никакого машинного обучения — чистый исчерпывающий перебор с полной верификацией эквивалентности состояний.

Один запуск на Apple M2 (3 ч 16 мин, 34,7 миллиарда сравнений) выдаёт **602 008 доказуемо корректных правил оптимизации по 83 уникальным паттернам трансформаций**. Примеры: SLA A : RR A → OR A (экономия 3 байта, 12T); LD A, 0 : NEG → SUB A (экономия 2 байта); SCF : RR A → SCF : RRA (экономия 1 байт, 4T). Важно: он корректно *отвергает* LD A, 0 → XOR A, потому что поведение флагов различается — та самая тонкая разница, которую вручную поддерживающие таблицы подглядывания иногда упускают.

Полезен как пост-обработка для сгенерированного компилятором кода Z80 или как справочник для ручной оптимизации. Результат — машиночитаемая база правил, которую можно интегрировать в инструменты ассемблера. См. Главу 23 об обсуждении брутфорс-подходов vs нейросетевых подходов к оптимизации Z80.

Исходники: github.com/oisee/z80-optimizer (лицензия MIT, v0.1.0)

Общая философия

Все эти инструменты разделяют принцип, напрямую переносимый на работу с Z80: **процедурная генерация побеждает хранимые данные**. На РС это означает генерацию текстур из функций шума вместо хранения растровых

изображений. На Spectrum это означает вычисление таблицы синусов из параболической аппроксимации вместо хранения 256 байт предвычисленных значений. Аппаратура различается в миллион раз; подход тот же.

J.6 Музыкальные и звуковые инструменты

PC-демосцена нуждается в крошечных синтезаторах — программных инструментах, помещающихся в 4К- или 64К-исполнимый файл и при этом производящих музыку профессионального качества. Z80-сцена нуждается в трекерах для AY-3-8910. Есть некоторое пересечение.

Sointu

Sointu — синтезатор для 4К-интро — форк 4klang, переписанный на Go для кроссплатформенности. Он предоставляет VST-подобный интерфейс для проектирования патчей (осцилляторы, фильтры, огибающие, эффекты) и компилятор, генерирующий минимальный нативный проигрыватель.

Код проигрывателя спроектирован для экстремально малого размера: весь рантайм синтезатора, включая все определения инструментов и данные нот, помещается менее чем в 4 КБ кода + данных x86. Это делает его PC-эквивалентом того, чего AY-beat (Глава 13) достигает на Spectrum: максимум звука из минимума байтов.

Исходники: github.com/vsariola/sointu (лицензия MIT)

4klang

4klang (автор Alcatraz) — оригинальный синтезатор для 4К-интро, от которого был форкнут Sointu. Он предоставляет VSTi-плагин для сочинения и режим вывода ассемблера, генерирующий NASM-исходник для проигрывателя — синтезатор буквально написан на ассемблере x86, оптимизированном по размеру.

4klang определил стандарт музыки для 4К-интро и по-прежнему активно используется наряду со своим наследником Sointu.

Исходники: github.com/gopher-atz/4klang

WaveSabre

WaveSabre (автор Logicoma) нацелен на 64К-интро — более крупный размерный класс, где синтезатор может быть сложнее. Он предоставляет VST-совместимые инструменты с реверберацией, задержкой, хорусом, дисторшном и другими эффектами, которые 4К-синтезаторы вынуждены опускать. Рантайм-проигрыватель достаточно компактен для 64К, но слишком велик для 4K.

WaveSabre стоит за музыкой в отмеченных наградами 64К-интро Logicoma (Happy Coding, Elysian).

Исходники: github.com/logicomacorp/WaveSabre (лицензия MIT)

Oidos

Oidos (автор Blueberry / Loonies) использует принципиально иной подход: аддитивный синтез. Там, где большинство синтезаторов строят звук из осцилляторов и фильтров, Oidos строит его из сумм синусоидальных волн с индивидуально управляемыми частотами и амплитудами. Результат — характерный, богатый звук, занимающий уникальное звуковое пространство в 4К-интро.

Исходники: github.com/askeksha/Oidos

Furnace

Furnace — мультисистемный чиптюн-трекер, поддерживающий более 80 звуковых чипов — включая **AY-3-8910**. Это делает его непосредственно актуальным для создания демо на ZX Spectrum: ты можешь сочинять музыку для AY в Furnace с современным интерфейсом, имеющим возможности, которых нет в Vortex Tracker II (отмена, несколько вкладок, визуальный редактор огибающей, осциллограф по каналам).

Среди других поддерживаемых чипов: SN76489 (Sega Master System, BBC Micro), YM2612 (Mega Drive), SID (C64), Pokey (Atari) и многие другие. Если ты работаешь над мультиплатформенными ретро-проектами, Furnace — единственный трекер, покрывающий все цели.

Форматы экспорта. Furnace может экспортировать в формат VGM (Video Game Music), который захватывает сырье записи в регистры по кадрам. Для AY-3-8910 это означает поток 14-байтных дампов регистров при 50/60 Гц — напрямую используемый на Spectrum с минимальным проигрывателем, записывающим регистры на каждом прерывании. Пользовательские скрипты экспорта могут конвертировать VGM в PT3 или в сырье таблицы регистров для INCBIN.

Ограничение. Эмуляция AY в Furnace хороша, но не идентична воспроизведению PT3 в Vortex Tracker. Если твоё демо использует PT3-проигрыватель, сочниай в Vortex Tracker для гарантированного воспроизведения 1:1. Используй Furnace, когда хочешь современный опыт редактирования и готов заниматься конвертацией форматов, или когда работаешь с AY напрямую через сырье записи в регистры.

Исходники: github.com/tildearrow/furnace (GPL-2.0)

Сравнительная таблица

Инструмент	Размерный класс	Лицензия	Актуальность для Z80
Sointu	4К-интро	MIT	Концептуальная (те же ограничения, что и AY-beat)
4klang	4К-интро	OSS	Концептуальная
WaveSabre	64К-интро	MIT	Концептуальная
Oidos	4К-интро	OSS	Концептуальная
Furnace	Любой	GPL-2.0	Прямая — поддержка AY-3-8910, экспорт VGM

Инструмент	Размерный класс	Лицензия	Актуальность для Z80
Vortex Tracker II	Любой	Freeware	Прямая — нативный PT3, стандарт ZX

J.7 Практические рецепты

Пять пошаговых рабочих процессов, связывающих современные инструменты с ассемблером Z80. Каждый рецепт начинается с пустого проекта и заканчивается данными, которые можно подключить в демо через INCBIN.

Рецепт 1: GNU Rocket → таблица синхронизации Z80

Цель: Спроектировать интерполированные кривые синхронизации в Rocket, экспортить как таблицы Z80 dw.

Шаги:

- Установи Rocket.** Клонируй github.com/rocket/rocket, собери из исходников (CMake). Запусти редактор.
- Создай треки.** Добавь четыре трека: effect:id, fade:alpha, scroll:speed, flash:border. Установи BPM в соответствии с музыкой (например, 125 BPM при 50fps = 24 кадра на долю).
- Расставь ключевые кадры.** В строке 0: effect:id = 0 (logo), fade:alpha = 255. В строке 150: effect:id = 1 (plasma), fade:alpha плавно меняется от 255 к 0 (smooth-интерполяция). Продолжи для всей длительности демо.
- Экспортируй.** Используй экспортер синхронизации Rocket (или утилиту sync_export) для записи бинарных файлов по каждому треку.
- Конвертируй с помощью Python:**

```
def rocket_to_z80(track_file, output_file):
    with open(track_file, 'rb') as f:
        data = f.read()
    # Rocket binary: array of (row: u32, value: float32) pairs
    pairs = []
    for i in range(0, len(data), 8):
        row, val = struct.unpack('<If', data[i:i+8])
        pairs.append((row, max(0, min(255, int(val)))))

    with open(output_file, 'w') as f:
        for row, val in pairs:
            f.write(f"    dw {row}, {val}\n")
        f.write("    dw 0 ; end marker\n")

rocket_to_z80('effect_id.track', 'sync_effect.inc')
```

- Подключи в демо:**

```
INCLUDE "sync_effect.inc"
```

Вариант с запеканием. Для гладкой интерполяции без затрат в рантайме разверни ключевые кадры в покадровые значения:

```
for frame in range(total_frames):
    value = interpolate(keyframes, frame) # linear, smooth, or ramp
    print(f"    db {max(0, min(255, int(value)))}")
```

Трек на 3 000 кадров, запечённый в покадровые значения db = 3 000 байт без сжатия, ~800 байт после ZX0.

Рецепт 2: Blender VSE → таблица номеров кадров

Цель: Визуально спланировать структуру демо с музыкой, экспортить точки синхронизации.

Шаги:

- Создай проект Blender.** Установи частоту кадров 50fps (Properties → Output → Frame Rate → 50). Установи длину таймлайна под свою музыку.
- Импортируй музыку.** Add → Sound в VSE. Импортируй свой .pt3, экспортированный в .wav (используй «File → Export to WAV» в Vortex Tracker). Волновая форма появится на таймлайне.
- Добавь полосы эффектов.** Add → Color strips для каждого эффекта. Назови их (plasma, scroll, torus). Задай цвета. Расположи на таймлайне так, чтобы они покрывали всю длительность демо.
- Расставь маркеры.** Перематывай таймлайн. Когда слышишь удар или точку перехода, нажми M, чтобы добавить маркер. Переименуй каждый маркер: plasma_start, flash_1, scroll_begin и т.д.
- Экспортируй через Python-консоль:**

```
# Print sync table
print("; Auto-generated sync table from Blender VSE")
print("sync_table:")
for m in sorted(bpy.context.scene.timeline_markers,
                key=lambda x: x.frame):
    print(f"    dw {m.frame} ; {m.name}")
print("    dw 0 ; end marker")
```

- Скопируй вывод в .inc-файл проекта.** Пересобери демо с помощью make.
- Итерируй.** Посмотри демо, отметь, где тайминг ощущается неправильно, вернись в Blender, скорректируй маркеры, переэкспортуй. Каждая итерация занимает секунды.

Рецепт 3: Unity VR → данные траектории

Цель: Захватить нарисованную вручную 3D-траекторию с помощью VR-контроллера, экспортить как сжатые данные для Z80.

Шаги:

- 1. Настрой проект Unity.** Создай новый 3D-проект, установи XR Toolkit через Package Manager. Настрой под свой шлем (Quest, Index и т.д.).
- 2. Запиши траекторию контроллера.** Создай скрипт, захватывающий transform.position правого контроллера каждый кадр:

```

positions.Add(new Vector3(
    controller.transform.position.x,
    controller.transform.position.y,
    controller.transform.position.z
));
}

```

- 3. Экспортируй CSV.** По окончании записи запиши позиции в CSV:

```
positions.Select(p => $"{p.x},{p.y},{p.z}");
```

- 4. Конвертируй с помощью Python:**

```

with open('trajectory.csv') as f:
    rows = list(csv.reader(f))

# Downsample from 90Hz to 50Hz
factor = 90 / 50
resampled = [rows[int(i * factor)] for i in range(int(len(rows) / factor))]

# Quantise to signed 8-bit (-128..127)
def q8(val, scale=64.0):
    return max(-128, min(127, int(float(val) * scale)))

# Delta-encode
prev = [0, 0, 0]
deltas = []
for row in resampled:
    cur = [q8(row[0]), q8(row[1]), q8(row[2])]
    deltas.append([c - p for c, p in zip(cur, prev)])
    prev = cur

# Transpose (column-major) and output
print("; X deltas")
print("traj_dx:")
print("    db " + ".join(str(d[0] & 0xFF) for d in deltas))
print("; Y deltas")
print("traj_dy:")
print("    db " + ".join(str(d[1] & 0xFF) for d in deltas))
print("; Z deltas")
print("traj_dz:")
print("    db " + ".join(str(d[2] & 0xFF) for d in deltas))

```

- 5. Сожми.** Пропусти каждый столбец через ZX0 отдельно (столбцовая раскладка сжимается намного лучше чередующейся). Подключи сжатые блоки через INCBIN.
- 6. Воспроизведение на Z80.** Распакуй каждый столбец в буфер. Каждый кадр читай следующую дельту и прибавляй к текущей позиции. Рисуй

полученную точку (X, Y) на экране (проецируй Z для глубины при необходимости).

Рецепт 4: Furnace → музыка для AY

Цель: Сочинить музыку для AY-3-8910 в Furnace и экспортовать для воспроизведения на ZX Spectrum.

Шаги:

1. **Настрой Furnace.** Создай новый проект. Добавь систему AY-3-8910 (Settings → выбери «AY-3-8910» из списка чипов). Установи тактовую частоту 1,7734 МГц (стандарт ZX Spectrum). Установи частоту обновления 50 Гц (PAL).
2. **Сочиняй.** Используй паттерн-редактор Furnace — похож на Vortex Tracker, но с отменой, осцилограммой по каналам и визуальным редактором огибающей. Furnace поддерживает аппаратные огибающие AY, микширование шума и все стандартные возможности AY.
3. **Экспортируй в VGM.** File → Export → VGM. Это создаёт .vgm-файл, содержащий сырье записи в регистры AY по кадрам — поток пар (регистр, значение) при 50 Гц.
4. **Конвертируй VGM в дамп регистров:**

```
def vgm_to_ay_regs(vgm_file):
    with open(vgm_file, 'rb') as f:
        data = f.read()

    # Skip VGM header (find data offset at 0x34)
    data_offset = struct.unpack_from('<I', data, 0x34)[0] + 0x34

    frames = []
    current_REGS = [0] * 14
    pos = data_offset

    while pos < len(data):
        cmd = data[pos]
        if cmd == 0xA0: # AY-3-8910 register write
            reg = data[pos + 1]
            val = data[pos + 2]
            if reg < 14:
                current_REGS[reg] = val
            pos += 3
        elif cmd == 0x62: # Wait 1/50s
            frames.append(list(current_REGS))
            pos += 1
        elif cmd == 0x66: # End of data
            break
        else:
            pos += 1

    return frames
```

```

frames = vgm_to_ay_regs('music.vgm')

# Output as Z80 include
print(f"music_frames: equ {len(frames)}")
print("music_data:")
for regs in frames:
    print("    db " + ", ".join(f"${r:02X}" for r in regs))

```

5. **Проигрыватель для Z80.** Простейший возможный проигрыватель — 14 инструкций OUT на прерывание:

```

ld hl, (music_ptr)
ld b, 14
xor a
.loop:
    out ($FD), a      ; select register
    ld c, (hl)
    inc hl
    push af
    ld a, c
    out ($BF), a      ; write value
    pop af
    inc a
    djnz .loop
    ld (music_ptr), hl
    ret

```

6. **Альтернатива: конвертация в PT3.** Если предпочитаешь стандартный PT3-проигрыватель (меньше, лучше сжатие), используй утилиту vgm2pt3 или сочиняй напрямую в Vortex Tracker. Преимущество Furnace — современный интерфейс; преимущество Vortex Tracker — гарантированная совместимость с PT3.

Рецепт 5: packbench → предварительный анализ перед сжатием

Цель: Проанализировать сырье данные демо перед сжатием, чтобы определить оптимальные преобразования.

Шаги:

- Собери без сжатия.** Собери демо с сырьими (несжатыми) данными INCBIN.
- Запусти packbench analyse** для каждого файла данных:

- Прочитай отчёт.** packbench сообщает:
 - **Энтропия** (бит на байт) — теоретический минимальный сжатый размер
 - **Распределение байтов** — показывает, группируются ли значения (хорошо) или распределены равномерно (плохо для сжатия)
 - **Длины серий** — показывает паттерны повторений
 - **Рекомендованные преобразования** — дельта-кодирование, разделение на битовые плоскости, транспонирование и т.д.

4. **Примени рекомендованные преобразования.** Если packbench предлагает дельта-кодирование:

```
deltas = bytes([(data[i] - data[i-1]) & 0xFF) for i in range(1, len(data))])
open('sprites_delta.bin', 'wb').write(bytes([data[0]]) + deltas)
```

5. **Повтори анализ.** Запусти packbench для преобразованных данных. Энтропия должна быть ниже.
6. **Сожми.** Пропусти преобразованные данные через ZX0 или Pletter. Сравни сжатый размер с оригиналом — преобразование должно дать меньший результат.
7. **Обнови демо.** Распаковщик Z80 запускается первым (декодирование ZX0), затем обратное преобразование (обратное дельта-кодирование) восстанавливает исходные данные. Обратное преобразование дёшево: ~10 T-states на байт для дельта-декодирования.
-

Дополнительное чтение

- **GNU Rocket:** github.com/rocket/rocket — редактор синхронизации + клиентские библиотеки
- **TiXL:** github.com/tixl3d/tixl — нодовая моушн-графика (MIT)
- **Архив Farbrausch:** github.com/farbrausch/fr_public — Werkzeug, kkrunchy, V2 (BSD)
- **Furnace:** github.com/tildearrow/furnace — мультисистемный чиптюн-трекер (GPL-2.0)
- **Sointu:** github.com/vsariola/sointu — синтезатор для 4K-интро (MIT)
- **WaveSabre:** github.com/logicomacorp/WaveSabre — синтезатор для 64K-интро (MIT)
- **Crinkler:** github.com/runestubbe/Crinkler — сжимающий линкер (zlib)
- **Bonzomatic:** github.com/Gargaj/Bonzomatic — живое шейдерное кодирование
- **Shader Minifier:** github.com/laurentlb/shader-minifier — оптимизатор GLSL/HLSL
- **z80-optimizer:** github.com/oisee/z80-optimizer — брутфорс-супероптимизатор Z80 (MIT)
- **Motion Canvas:** motioncanvas.io — параметрическая анимация (MIT)
- **Blender:** blender.org — 3D, VSE, Graph Editor, Geometry Nodes, Grease Pencil (GPL)